https://simplifycpp.org

DRAFT

Modern C++ Web Development with Wt Toolkit

Designing and Deploying Production-Grade Server-Side Applications



Prepared by Ayman Alheraki

Modern C++ Web Development with Wt Toolkit

Designing and Deploying Production-Grade Server-Side Applications

Some drafting assistance and idea exploration were supported by modern AI tools, with full supervision, verification, correction, and authorship.

Prepared by Ayman Alheraki

Contents

Author's Introduction **52**

A short note on code style and toolchains 55

A minimal conceptual sketch 55

Book Preface **57**

What this book is 57

What this book is not 58

Audience and prerequisites 59

The learning method used in this book 59

The live project: scope and rationale 60

Architectural rules enforced by the book 60

A note on security, correctness, and "most recent" practice 61

How to use this book 61

A minimal build-and-run checkpoint 62

I Conceptual and Technical Foundations 63

1 Why Wt? Why C++ on the Web? 64

1.1 Why Wt was created 64

1.1.1 The engineering gap Wt addresses 64
1.1.2 The central idea: server-side stateful interaction 65
1.1.3 A deliberate alternative to client-heavy stacks 65
1.1.4 Type safety and explicitness as first-class goals 66
1.1.5 The UI-as-widgets model and event-driven programming 67
1.1.6 Operational clarity: the application as a long-running service 68
1.1.7 Where Wt is most justified 69
1.1.8 Design tradeoffs and the boundaries of the approach 69
1.1.9 Summary 70
1.2 The fundamental difference between Wt and JavaScript frameworks 70
1.2.1 A difference of architecture, not syntax 70
1.2.2 Where the “truth” lives: state ownership and authority 71
1.2.3 Programming model: event-driven server objects vs client components 72
1.2.4 The toolchain boundary: one runtime vs two runtimes 74
1.2.5 Statefulness, sessions, and scalability assumptions 75
1.2.6 Security boundaries and trust distribution 75
1.2.7 User experience: responsiveness vs round-trips 76
1.2.8 Maintainability: consistency and architectural gravity 76
1.2.9 Summary 77
1.3 When Wt is the right choice — and when it is not 77
1.3.1 The decision is architectural, not ideological 77
1.3.2 A practical checklist for “right choice” 78
1.3.3 Typical domains where Wt is a strong fit 80
1.3.4 When Wt is not the right choice 81
1.3.5 Decision framework: choose the model that reduces total system risk . 83
1.3.6 A concise rule of thumb 83
1.3.7 Summary 83

1.4 Wt as a server-side web platform 84
1.4.1 Platform meaning: more than a UI library 84
1.4.2 The hosting model: a long-running C++ service 85
1.4.3 Sessions as first-class runtime entities 86
1.4.4 The widget tree as an application model 86
1.4.5 Event-driven interaction: browser events to C++ handlers 87
1.4.6 Transport and synchronization: incremental updates 88
1.4.7 Resources and theming: integrated web concerns 88
1.4.8 Integration with the C++ ecosystem 89
1.4.9 Operational deployment: reverse proxy and service management 89
1.4.10 A platform-oriented architecture: layered responsibility 90
1.4.11 Summary 90
1.5 Real-world use cases (Enterprise, Government, Internal Systems) 91
1.5.1 Why these domains align with Wt 91
1.5.2 Enterprise applications 92
1.5.3 Government and institutional systems 93
1.5.4 Internal systems and private platforms 95
1.5.5 Characteristics that repeatedly appear in successful Wt deployments . . 97
1.5.6 Where these use cases fail if the architecture is wrong 97
1.5.7 A small concrete scenario: role-based administrative workflow 98
1.5.8 Summary 99

2 How Wt Works Internally **100**
2.1 Application lifecycle 100
2.1.1 Overview 100
2.1.2 Process startup and configuration 100
2.1.3 Session creation and request processing 101
2.1.4 Widget tree initialization 102

2.1.5 Event handling and user interaction . . . 103
2.1.6 Session persistence, timeout, and cleanup . . . 104
2.1.7 Handling long-running processes and concurrency . . . 105
2.1.8 Application shutdown and cleanup . . . 105
2.1.9 Lifecycle observability and diagnostics . . . 106
2.1.10 Summary . . . 106
2.2 One `WApplication` per session . . . 107
2.2.1 The central runtime model . . . 107
2.2.2 What "session" means in Wt . . . 107
2.2.3 How the framework creates a `WApplication` instance . . . 108
2.2.4 Consequences: isolation, state, and correctness . . . 109
2.2.5 Widget tree ownership and lifetime . . . 110
2.2.6 Shared state vs session state: what must not be confused . . . 110
2.2.7 Thread-safety implications . . . 111
2.2.8 Why this model changes how you design your architecture . . . 112
2.2.9 A small, concrete pattern: session context . . . 113
2.2.10 Memory and capacity planning . . . 114
2.2.11 Termination and deterministic cleanup . . . 114
2.2.12 Summary . . . 115
2.3 Event loop and request handling model . . . 115
2.3.1 Introduction . . . 115
2.3.2 The server as an event-driven runtime . . . 115
2.3.3 Connection handling: HTTP and WebSockets . . . 116
2.3.4 The central event loop . . . 117
2.3.5 From request to session dispatch . . . 118
2.3.6 Widget event handling . . . 119
2.3.7 Incremental UI updates . . . 119

2.3.8 Timers and internal events 120
2.3.9 Asynchronous operations 120
2.3.10 Concurrency and thread management 121
2.3.11 Graceful shutdown and event loop termination 121
2.3.12 Summary 122
2.4 Stateful web applications 122
2.4.1 What “stateful” means in the context of Wt 122
2.4.2 Why state matters in real systems 123
2.4.3 The Wt mechanism: per-session application instances 124
2.4.4 Kinds of state in a stateful Wt application 124
2.4.5 The benefits of statefulness in Wt 126
2.4.6 The costs and risks of statefulness 127
2.4.7 A disciplined pattern: explicit session context 128
2.4.8 A disciplined pattern: keep persistent state persistent 129
2.4.9 A small example: stateful multi-step workflow 129
2.4.10 Operational implications: timeouts, cleanup, and bounded state 130
2.4.11 Summary 131
2.5 HTTP vs WebSockets in Wt 131
2.5.1 Introduction 131
2.5.2 The HTTP request/response model 132
2.5.3 HTTP message lifecycle in Wt 132
2.5.4 Limitations and overhead of HTTP-only interaction 133
2.5.5 WebSocket protocol: persistent and bi-directional 133
2.5.6 WebSocket handshake and integration in Wt 134
2.5.7 How Wt chooses the transport 134
2.5.8 Event transport and synchronization 135
2.5.9 Incremental UI updates over WebSocket 135

2.5.10 Production considerations: fallback, proxies, and compatibility 136
2.5.11 Performance implications 136
2.5.12 Security considerations 136
2.5.13 API implications for application developers 137
2.5.14 Monitoring and diagnostics 137
2.5.15 Fallback and progressive enhancement 138
2.5.16 Summary 138

II Environment Setup and First Application 139

3 Development Environment Setup 140
3.1 Requirements on Linux and Windows 140
3.1.1 What you must have before you start 140
3.1.2 Minimum versions (portable baseline) 140
3.1.3 Linux requirements 141
3.1.4 Windows requirements 143
3.1.5 Practical guidance: choose your dependency profile 145
3.1.6 Summary 145
3.2 Installing Wt (packages vs source) 146
3.2.1 Two professional installation strategies 146
3.2.2 Option A: Install from packages (fast, simple, system-integrated) 146
3.2.3 Option B: Build from source (control, reproducibility, production alignment) 148
3.2.4 Choosing between packages and source: a decision matrix 151
3.2.5 Book recommendation: what this book should assume 152
3.3 CMake configuration for Wt projects 152
3.3.1 Why CMake is the professional baseline for Wt 152

3.3.2 Target-based linking: the core rule . . . 153
3.3.3 Minimal Wt application: canonical CMake structure . . . 153
3.3.4 Build directory discipline (out-of-source builds) . . . 154
3.3.5 When CMake cannot find Wt . . . 155
3.3.6 Selecting the right Wt components . . . 156
3.3.7 RPATH and runtime library discovery (Linux) . . . 156
3.3.8 A scalable CMake layout for real projects . . . 157
3.3.9 A clean multi-target example . . . 158
3.3.10 Summary . . . 159
3.4 Establishing a clean project structure from day one . . . 159
3.4.1 Why structure matters more in Wt than in toy web stacks . . . 159
3.4.2 Design goals for the project layout . . . 160
3.4.3 Layers that should exist early . . . 160
3.4.4 A production-friendly directory layout . . . 162
3.4.5 Core principle: keep `WApplication` minimal . . . 164
3.4.6 A clean separation example: UI event to service call . . . 164
3.4.7 CMake structure: libraries first, executable second . . . 167
3.4.8 Configuration and deployment assets: treat them as first-class . . . 168
3.4.9 Testing readiness from the beginning . . . 168
3.4.10 A disciplined growth path . . . 169
3.4.11 Summary . . . 169

4 Your First Wt Application (Hello Wt) 170
4.1 Analyzing a minimal example . . . 170
4.1.1 Purpose of a minimal example . . . 170
4.1.2 The code: a minimal Wt application . . . 170
4.1.3 Include directives and core classes . . . 171
4.1.4 Defining the application class . . . 172

4.1.5 The server entry point . . . 173
4.1.6 Registering the application entry point . . . 173
4.1.7 Running the server . . . 174
4.1.8 What happens at runtime . . . 175
4.1.9 Extending minimal examples . . . 175
4.1.10 A professional note on configuration . . . 176
4.1.11 Summary . . . 176
4.2 Understanding `main()` and `WServer` . . . 177
4.2.1 Why `main()` is part of the framework contract . . . 177
4.2.2 What `Wt::WServer` is . . . 177
4.2.3 The canonical `main()` structure . . . 178
4.2.4 Understanding the `WServer` constructor . . . 179
4.2.5 Entry points: what you are really registering . . . 180
4.2.6 What happens when `run()` is called . . . 181
4.2.7 Request routing and session lookup (conceptual) . . . 181
4.2.8 Threading and concurrency at the `WServer` level . . . 182
4.2.9 A production pattern: wiring shared services into sessions . . . 182
4.2.10 Common mistakes and the professional correction . . . 183
4.2.11 Summary . . . 184
4.3 Entry points explained . . . 184
4.3.1 What an entry point represents . . . 184
4.3.2 Why entry points matter for modular design . . . 185
4.3.3 Types of entry points . . . 185
4.3.4 Standard application entry point . . . 186
4.3.5 How entry points support multiple behaviors . . . 186
4.3.6 How the server routes requests to entry points . . . 187
4.3.7 Factory semantics: why unique_ptr matters . . . 187

4.3.8 Example: multi-entry-point configuration 188
4.3.9 Entry points and session lifetime 189
4.3.10 Entry points and application configuration 189
4.3.11 Design patterns for entry point factories 190
4.3.12 Summary 190
4.4 Running the application locally 191
4.4.1 What “running locally” means for Wt 191
4.4.2 Prerequisites before the first run 191
4.4.3 Local run on Linux 192
4.4.4 Local run on Windows 193
4.4.5 Controlling the local server port and configuration 194
4.4.6 How to interpret the first run 194
4.4.7 Troubleshooting the most common local issues 195
4.4.8 Recommended local workflow for this book 197
4.4.9 Summary 197
4.5 First widget and root container 197
4.5.1 What the root container represents 197
4.5.2 Accessing the root container 198
4.5.3 Widget ownership and lifetime 198
4.5.4 Your first widget: `Wt::WText` 199
4.5.5 Why everything starts at the root 199
4.5.6 Using layout containers under the root 200
4.5.7 Event-driven behavior from the root 200
4.5.8 Clearing and replacing the root content 201
4.5.9 Styling the root container 201
4.5.10 Root container and session isolation 202
4.5.11 Summary 202

III Professional Project Architecture 203

5 Designing a Production-Ready Architecture 204

5.1 Layer separation (UI / Controllers / Services / Models) 204

5.1.1 Why layering is non-negotiable in production Wt systems 204

5.1.2 The four-layer contract 205

5.1.3 Layer 1: UI layer (Views / Widgets / Pages) 205

5.1.4 Layer 2: Controller layer (workflow orchestration) 206

5.1.5 Layer 3: Service layer (business rules and use cases) 207

5.1.6 Layer 4: Models (domain concepts and contracts) 208

5.1.7 Dependency direction: the single rule that prevents chaos 209

5.1.8 A minimal example illustrating the separation 209

5.1.9 Where persistence fits (repositories as a boundary) 213

5.1.10 Why this separation is especially important in Wt 213

5.1.11 Summary 214

5.2 Why classic MVC does not directly fit Wt 214

5.2.1 The problem is not MVC itself, but what “classic MVC” assumes . . . 214

5.2.2 Wt’s fundamental architecture: sessions and widget trees 215

5.2.3 Where classic MVC breaks 215

5.2.4 What replaces classic MVC in Wt: a better mapping 217

5.2.5 The key architectural rule: keep Wt types out of domain logic 218

5.2.6 A concrete example of the MVC trap 218

5.2.7 A Wt-aligned solution: controller + service boundary 219

5.2.8 Why this matters operationally 220

5.2.9 Summary 220

5.3 A scalable, project-driven architecture 221

5.3.1 Why “project-driven” is the only architecture that survives 221

5.3.2 Design goals for a scalable Wt architecture 221

5.3.3 A minimal set of modules that scale . . . 222
5.3.4 The dependency rule that prevents architectural collapse . . . 223
5.3.5 A directory layout that supports scaling . . . 223
5.3.6 Composition root: the architecture hinge in Wt . . . 225
5.3.7 A disciplined session model . . . 225
5.3.8 Use cases as the unit of growth . . . 226
5.3.9 An example: use-case centered structure . . . 226
5.3.10 CMake: enforce modules as targets . . . 228
5.3.11 Operational readiness as a first-class architectural concern . . . 229
5.3.12 Summary . . . 229
5.4 File and directory organization . . . 230
5.4.1 Purpose of disciplined organization . . . 230
5.4.2 Top-level project structure . . . 230
5.4.3 Build system support (`CMakeLists.txt` and `cmake/`) . . . 231
5.4.4 Configuration directory (`config/`) . . . 231
5.4.5 Static assets (`assets/`) . . . 232
5.4.6 Source directory (`src/`) . . . 232
5.4.7 Application entry point (`src/app/`) . . . 233
5.4.8 UI layer (`src/ui/`) . . . 233
5.4.9 Controller layer (`src/controllers/`) . . . 234
5.4.10 Service layer (`src/services/`) . . . 234
5.4.11 Domain layer (`src/domain/`) . . . 235
5.4.12 Data access layer (`src/data/`) . . . 235
5.4.13 Infrastructure layer (`src/infra/`) . . . 236
5.4.14 Tests directory (`tests/`) . . . 236
5.4.15 Scripts and automation (`scripts/`) . . . 236
5.4.16 Documentation (`docs/`) . . . 237

5.4.17 Summary 237

6 The Application Class in Depth **238**

6.1 Responsibilities of `WApplication` 238

6.1.1 What `WApplication` is in the Wt execution model 238

6.1.2 Primary responsibilities 239

6.1.3 Secondary responsibilities that appear in real systems 244

6.1.4 What `WApplication` should *not* do 245

6.1.5 A production template: thin `WApplication`, explicit router, explicit services 246

6.1.6 Summary 247

6.2 Themes and CSS integration 248

6.2.1 Why theming matters in a Wt production system 248

6.2.2 The two styling layers in Wt 248

6.2.3 Theme usage as an application-level responsibility 249

6.2.4 Setting a theme in `WApplication` 249

6.2.5 Adding stylesheets (CSS integration) 250

6.2.6 Organizing CSS for long-term scalability 251

6.2.7 Mapping widgets to CSS classes 252

6.2.8 A minimal CSS baseline for Wt applications 252

6.2.9 Theme-friendly component design 253

6.2.10 Serving CSS assets in development and production 253

6.2.11 Caching and versioning strategy 254

6.2.12 Accessibility and readability as first-class styling requirements 255

6.2.13 A professional pattern: a UI theme module 255

6.2.14 Summary 256

6.3 Session management 256

6.3.1 Why session management is a core architectural topic in Wt 256

6.3.2 The Wt session model in one sentence 257
6.3.3 What constitutes “session state” in a production Wt application 257
6.3.4 Session lifetime: creation, activity, expiration, teardown 258
6.3.5 Where to implement session policy: configuration vs code 259
6.3.6 A session context object: the minimal professional baseline 259
6.3.7 Authentication lifecycle: login, session binding, logout 260
6.3.8 Authorization: never trust UI-only protection 262
6.3.9 Concurrency and session safety 263
6.3.10 Scaling to multiple server instances 264
6.3.11 Memory discipline: the production survival rule 265
6.3.12 A minimal “session-aware” application skeleton 265
6.3.13 Summary 268
6.4 Working with environment and request data 268
6.4.1 What “environment and request data” means in Wt 268
6.4.2 The role of `Wt::WEnvironment` 269
6.4.3 Accessing environment data during construction 269
6.4.4 Query parameters and navigation context 270
6.4.5 Cookies: reading and writing 270
6.4.6 Headers and client metadata 271
6.4.7 Remote address and transport information 272
6.4.8 Form data and widget-bound inputs 272
6.4.9 URL fragments and internal navigation 272
6.4.10 Handling POST/PUT/DELETE semantics 273
6.4.11 Security considerations when using environment data 273
6.4.12 Localization and request context 274
6.4.13 A session-aware request abstraction 274
6.4.14 Summary 275

IV Building the User Interface (UI Layer) 276

7 Core Wt Widgets 277

7.1 Containers . . . 277
7.1.1 What a container means in Wt . . . 277
7.1.2 Core ownership rule: containers own their children . . . 277
7.1.3 The fundamental container: `WContainerWidget` . . . 278
7.1.4 Layout-based containers: structuring space and alignment . . . 279
7.1.5 Structured containers: common UI building blocks . . . 281
7.1.6 Template containers: separating structure from content . . . 283
7.1.7 Performance and scalability considerations . . . 284
7.1.8 Testing and maintainability guidelines . . . 284
7.1.9 Summary . . . 285
7.2 Input widgets . . . 285
7.2.1 Why input widgets are a production-critical topic . . . 285
7.2.2 Core principles for input handling in Wt . . . 286
7.2.3 Text input: `WLineEdit` . . . 286
7.2.4 Multi-line input: `WTextArea` . . . 288
7.2.5 Numeric and typed input: `WSpinBox`, `WDoubleSpinBox` . . . 288
7.2.6 Boolean input: `WCheckBox` . . . 289
7.2.7 Single choice: `WRadioButton` and `WButtonGroup` . . . 290
7.2.8 Drop-down selection: `WComboBox` . . . 290
7.2.9 Date and time input . . . 291
7.2.10 File upload: `WFileUpload` . . . 291
7.2.11 Validation: widgets + validators . . . 292
7.2.12 Form handling: consistent boundary validation . . . 293
7.2.13 Security discipline for input widgets . . . 294
7.2.14 Maintainability: building reusable field components . . . 294

7.2.15 Summary 294
7.3 Buttons 295
7.3.1 Why buttons deserve serious engineering attention 295
7.3.2 Core button widgets in Wt 295
7.3.3 `WPushButton`: the standard command button 296
7.3.4 Disabling buttons and preventing double-submit 297
7.3.5 Default buttons and keyboard interaction 297
7.3.6 Buttons as navigation vs buttons as commands 298
7.3.7 Styling buttons with CSS classes 298
7.3.8 Icon buttons and compact controls 299
7.3.9 Confirmation for destructive actions 299
7.3.10 Progress feedback and long operations 301
7.3.11 Button-driven form validation workflow 301
7.3.12 Maintainability: reusable button patterns 302
7.3.13 Summary 302
7.4 Layouts 303
7.4.1 Why layouts are the real foundation of professional Wt UIs 303
7.4.2 Two layout approaches in Wt 303
7.4.3 The layout ownership rule 304
7.4.4 Vertical layout: `WVBoxLayout` 304
7.4.5 Horizontal layout: `WHBoxLayout` 305
7.4.6 Grid layout: `WGridLayout` 306
7.4.7 Border layout: `WBorderLayout` 307
7.4.8 Stacked layouts: view switching without rebuilding 308
7.4.9 Layout + CSS: the professional hybrid approach 309
7.4.10 Responsive design strategy 310
7.4.11 Performance and maintainability guidelines 311

7.4.12 Summary 311
7.5 Events and signals 311
7.5.1 Why the event model is the core of Wt 311
7.5.2 Signals in Wt: the conceptual model 312
7.5.3 The basic pattern: connect a handler 312
7.5.4 Signal categories you use constantly 314
7.5.5 Event dispatch and session context 315
7.5.6 Avoiding the most common production mistake: lifetime capture bugs . 316
7.5.7 Signal design: keep handlers thin 317
7.5.8 Chaining events and composing workflows 318
7.5.9 Asynchronous work and signals: safe UI updates 318
7.5.10 Event storms and performance 319
7.5.11 Event and signal patterns you should standardize 320
7.5.12 Summary 320

8 Building a Professional Application Layout 321
8.1 Header, sidebar, and content areas 321
8.1.1 Why structured page areas matter in professional UIs 321
8.1.2 The structural pattern: application shell 321
8.1.3 Header area 322
8.1.4 Sidebar area 323
8.1.5 Content area 324
8.1.6 Putting it all together: a complete shell example 324
8.1.7 CSS integration for layout regions 326
8.1.8 Responsive and collapsible sidebar patterns 327
8.1.9 Accessibility and keyboard navigation 328
8.1.10 Summary 328
8.2 Reusable UI components 329

8.2.1 Why reuse is mandatory in production UI code 329
8.2.2 The three layers of reuse in Wt UI architecture 330
8.2.3 Component design rules (production discipline) 331
8.2.4 Pattern: atomic reusable component (`LabeledField`) 331
8.2.5 Pattern: reusable button builders (primary, secondary, danger) 333
8.2.6 Pattern: composite reusable component (`FormSection`) 334
8.2.7 Pattern: page components with a narrow API 335
8.2.8 Event wiring inside components: avoid leaking internals 337
8.2.9 Styling reusable components: semantic CSS contract 338
8.2.10 Reusable components and validation policy 338
8.2.11 Reusable components and performance 339
8.2.12 Recommended component directory organization 339
8.2.13 Summary 340
8.3 Dynamic UI updates 340
8.3.1 What “dynamic UI updates” means in Wt 340
8.3.2 The fundamental update pipeline 341
8.3.3 Immediate updates from event handlers 341
8.3.4 Updating subtrees: region replacement vs incremental change 343
8.3.5 Dynamic list/table updates with models 344
8.3.6 Timers: periodic dynamic updates 344
8.3.7 Debounced updates (live search and reactive filters) 345
8.3.8 Long-running work: asynchronous updates and busy states 346
8.3.9 Server push: updates initiated by the server 347
8.3.10 Partial rendering strategy: update small regions 348
8.3.11 Consistency rules for dynamic UI updates 348
8.3.12 Common pitfalls and how to avoid them 349
8.3.13 Summary 349

8.4 Page navigation management 349
8.4.1 Why navigation is an architecture problem in Wt 349
8.4.2 The professional layout assumption: stable shell + replaceable content . 350
8.4.3 The three navigation responsibilities 351
8.4.4 Navigation mechanisms in Wt 351
8.4.5 Internal path as the canonical navigation state 352
8.4.6 A minimal router for production use 352
8.4.7 Integrating navigation with header and sidebar 355
8.4.8 Stacked navigation: when it is appropriate 356
8.4.9 Deep links and parameterized routes 356
8.4.10 Navigation and authorization: do not rely on hidden links 357
8.4.11 Navigation and session lifetime 358
8.4.12 Navigation and UX consistency: active state and breadcrumbs 358
8.4.13 Error handling routes 359
8.4.14 Recommended directory organization for navigation code 359
8.4.15 Summary 359

V The Live Project: User Management System 361

9 Designing the User System 362
9.1 System requirements 362
9.1.1 Target deployment profiles 362
9.1.2 Operating systems and runtime environment 363
9.1.3 Compiler and language standard 363
9.1.4 Build system and tooling 364
9.1.5 Core Wt components required by the project 365
9.1.6 Database requirements 366

9.1.7 Security and cryptography requirements . . . 367
9.1.8 Web server / connector requirements . . . 368
9.1.9 Email and notification requirements . . . 369
9.1.10 Storage requirements . . . 370
9.1.11 Performance and scalability requirements . . . 370
9.1.12 Observability requirements . . . 371
9.1.13 Testing and quality gates . . . 371
9.1.14 Recommended production hardening checklist . . . 372
9.1.15 Summary . . . 372
9.2 Use cases . . . 373
9.2.1 Actors . . . 373
9.2.2 Account lifecycle use cases . . . 374
9.2.3 Authentication and session use cases . . . 375
9.2.4 Password reset and recovery use cases . . . 377
9.2.5 Authorization and role management use cases . . . 378
9.2.6 Multi-factor authentication (optional but common) . . . 379
9.2.7 Single Sign-On and external identity (optional) . . . 379
9.2.8 Operational and support use cases . . . 380
9.2.9 Wt-specific interaction patterns for use cases . . . 380
9.2.10 Acceptance criteria per use case (production baseline) . . . 383
9.2.11 Summary . . . 384
9.3 User, role, and permission modeling . . . 384
9.3.1 Design goals and constraints . . . 385
9.3.2 Core concepts and terminology . . . 386
9.3.3 Recommended RBAC model (baseline) . . . 387
9.3.4 Entity model (Wt::Dbo) . . . 388
9.3.5 Database constraints and invariants . . . 392

9.3.6 Permission evaluation strategy . . . 393
9.3.7 Record-level access control (ownership and scope) . . . 395
9.3.8 Role design guidelines (practical and maintainable) . . . 397
9.3.9 Permission catalog (recommended baseline for the live project) . . . 397
9.3.10 Auditing model (minimal requirements) . . . 399
9.3.11 Integration with authentication state in Wt . . . 399
9.3.12 Operational rules and safety checks . . . 401
9.3.13 Summary . . . 402

10 Login Page Implementation 403
10.1 Building the login view . . . 403
10.1.1 UI goals and interaction contract . . . 403
10.1.2 Recommended view structure . . . 404
10.1.3 Service interfaces used by the view . . . 404
10.1.4 Template-based login view implementation . . . 407
10.1.5 Integration: placing the login view in the application router . . . 416
10.1.6 Security requirements embedded in the login view . . . 418
10.1.7 UX improvements for production . . . 419
10.1.8 Minimal styling (optional but practical) . . . 420
10.1.9 Summary . . . 421
10.2 Input validation . . . 421
10.2.1 Validation objectives . . . 422
10.2.2 Two-layer validation model . . . 423
10.2.3 What to validate on a login form . . . 424
10.2.4 Wt validators: production usage pattern . . . 425
10.2.5 Safe feedback: what to show the user . . . 429
10.2.6 Avoiding common validation pitfalls . . . 430
10.2.7 Validation inside the authentication service . . . 431

10.2.8 Handling redirect targets safely (“next” path) 432
10.2.9 Testing the validators 433
10.2.10 Summary 434
10.3 Error handling 435
10.3.1 Error taxonomy for login flows 435
10.3.2 Security rules for login error handling 437
10.3.3 View-level error handling pattern 438
10.3.4 Service-level error handling and normalization 440
10.3.5 Login view: complete error-handling flow 442
10.3.6 Logging and auditing errors safely 445
10.3.7 Fail-safe behavior under backend outages 447
10.3.8 DoS-resistance considerations in error handling 447
10.3.9 Summary 448
10.4 Connecting the UI to the controller 448
10.4.1 Why a controller in a Wt project 449
10.4.2 Controller responsibilities for the login flow 450
10.4.3 Contracts: request, response, and outcomes 450
10.4.4 Service interfaces used by the controller 452
10.4.5 Controller implementation 455
10.4.6 Connecting Wt UI events to the controller 462
10.4.7 Router integration: passing the intended destination 464
10.4.8 Controller testing (without Wt) 465
10.4.9 Summary 467

11 Authentication and Session Management **468**
11.1 Login and logout flow 468
11.1.1 Key concepts: identity, authentication state, and session 469
11.1.2 Flow requirements 469

11.1.3 State machine: login/logout transitions 471
11.1.4 Capturing the intended destination (post-login redirect) 472
11.1.5 Router enforcement of authentication 473
11.1.6 Login flow implementation 474
11.1.7 Logout flow implementation 476
11.1.8 Session invalidation and forced logout 478
11.1.9 Recommended flow checks 479
11.1.10 Complete flow summary (sequence) 480
11.1.11 Summary 481
11.2 Session state handling 481
11.2.1 Session state goals 482
11.2.2 Wt session lifecycle 482
11.2.3 Session identity structure 483
11.2.4 Session versioning and forced logout 484
11.2.5 Session timeout policies 485
11.2.6 Cookie-based session identifier 487
11.2.7 Session storage considerations 487
11.2.8 Initialization and teardown 488
11.2.9 Periodic validation hooks 489
11.2.10 Session-based permission caching 490
11.2.11 Sensitive operations and re-authentication 490
11.2.12 Session cleanup and garbage collection 491
11.2.13 Testing session state logic 491
11.2.14 Summary 492
11.3 Protecting secured pages 492
11.3.1 Threat model for secured pages 493
11.3.2 Protection layers (defense in depth) 494

11.3.3 Central route guard using internal paths . . . 495
11.3.4 Continuous session validity checks . . . 497
11.3.5 Authorization checks: permissions and scope . . . 500
11.3.6 Router + page factory pattern . . . 501
11.3.7 UI-level protection (secondary layer) . . . 503
11.3.8 Protecting actions: never trust the UI . . . 504
11.3.9 CSRF and state-changing operations . . . 505
11.3.10 Production checklist for secured pages . . . 505
11.3.11 Summary . . . 506
11.4 User context management . . . 506
11.4.1 Design goals . . . 507
11.4.2 User context: what it contains . . . 508
11.4.3 C++ model: UserContext and PermissionCache . . . 509
11.4.4 Where the context lives in a Wt application . . . 510
11.4.5 Context provider pattern . . . 511
11.4.6 Integrating context provider into the Wt application . . . 517
11.4.7 Using user context in controllers and pages . . . 518
11.4.8 Context refresh and invalidation strategies . . . 520
11.4.9 Step-up authentication support . . . 521
11.4.10 Administrative impersonation (optional, high risk) . . . 522
11.4.11 Operational checklist . . . 523
11.4.12 Summary . . . 523

VI Database and Data Layer 525

12 Introduction to Wt::Dbo 526
12.1 ORM philosophy in Wt . . . 526

12.1.1 Core concepts in Wt::Dbo 527
12.1.2 Mapping philosophy 528
12.1.3 Transactions and identity 529
12.1.4 Lazy loading and eager fetch 529
12.1.5 Change tracking 530
12.1.6 Polymorphism support 530
12.1.7 Query mechanisms 531
12.1.8 Schema evolution and migrations 531
12.1.9 Error handling in ORM operations 532
12.1.10 Performance considerations 532
12.1.11 Testing and testability 533
12.1.12 Integration with application architecture 533
12.1.13 Summary 533
12.2 Sessions and transactions 534
12.2.1 The role of `dbo::Session` 535
12.2.2 The role of `dbo::Transaction` 536
12.2.3 Minimal example: creating a session and running a transaction 537
12.2.4 Why transactions matter even for reads 539
12.2.5 Identity map and object stability 539
12.2.6 Transactions and lazy loading 540
12.2.7 Nested transactions and composition 540
12.2.8 Unit of work pattern 541
12.2.9 Isolation levels and practical expectations 542
12.2.10 Error handling patterns 542
12.2.11 Concurrency and session usage constraints 543
12.2.12 Practical recipes for Wt applications 544
12.2.13 Summary 545

12.3 Object-relational mapping 546
12.3.1 The need for ORM 546
12.3.2 Mapping domain classes to tables 547
12.3.3 Mapping object fields 548
12.3.4 Mapping relationships 549
12.3.5 Lazy loading and explicit fetch control 551
12.3.6 Object identity and caching 551
12.3.7 Transactions and change tracking 552
12.3.8 Polymorphism and inheritance 552
12.3.9 Query API 552
12.3.10 Error handling in ORM operations 553
12.3.11 Schema evolution and migrations 554
12.3.12 Performance considerations 554
12.3.13 Separation of concerns in ORM 554
12.3.14 Testing with ORM 555
12.3.15 Summary 555

13 Data Modeling **556**
13.1 User model 556
13.1.1 Principles of a production user model 557
13.1.2 Schema design 559
13.1.3 C++ domain model and `Wt::Dbo` mapping 560
13.1.4 Mapping and constraints setup 564
13.1.5 User model behaviors used by authentication 564
13.1.6 Production recommendations 565
13.1.7 Summary 566
13.2 Role model 566
13.2.1 Role semantics and boundaries 567

13.2.2 Schema design . . . 568
13.2.3 Many-to-many relationships . . . 569
13.2.4 C++ domain model and `Wt::Dbo` mapping . . . 569
13.2.5 Mapping registration . . . 572
13.2.6 Role management rules (production policy) . . . 573
13.2.7 Practical modeling decisions . . . 574
13.2.8 Summary . . . 574
13.3 Relationships . . . 575
13.3.1 Relationship design principles . . . 575
13.3.2 Core relationships in the UMS . . . 577
13.3.3 Relationship implementation in `Wt::Dbo` . . . 579
13.3.4 Relationship correctness constraints (database-level) . . . 583
13.3.5 Performance and loading patterns . . . 584
13.3.6 Relationship-driven authorization model . . . 585
13.3.7 Operational rules for relationship updates . . . 585
13.3.8 Summary . . . 586
13.4 Lightweight migrations . . . 587
13.4.1 Why `createTables()` is not enough . . . 588
13.4.2 Definition: lightweight migrations . . . 588
13.4.3 Core pattern: migration ledger table . . . 589
13.4.4 Migration execution strategy . . . 589
13.4.5 Lightweight migration framework in C++ . . . 590
13.4.6 Writing migrations: schema + data . . . 595
13.4.7 Example migrations for the UMS . . . 596
13.4.8 Migration safety rules . . . 599
13.4.9 Recommended integration patterns . . . 600
13.4.10 Summary . . . 601

14 A Clean Database Layer **602**

14.1 Database singleton or connection pool 602

14.1.1 Connection basics and web application demands 602

14.1.2 Singleton connection pattern 603

14.1.3 Connection pool pattern 604

14.1.4 Implementing a connection pool in Modern C++ 606

14.1.5 Integration with Wt application sessions 609

14.1.6 Error and retry policies 610

14.1.7 Singleton vs pool: summary comparison 611

14.1.8 Advanced patterns: scoped sessions 611

14.1.9 Pooling in distributed environments 611

14.1.10 Testing and mocks 612

14.1.11 Summary 612

14.2 Separating SQL from business logic 613

14.2.1 The architectural goal 614

14.2.2 Layered model: the minimal clean separation 615

14.2.3 A common anti-pattern 615

14.2.4 Clean separation pattern 616

14.2.5 Repository design 616

14.2.6 Domain service design 621

14.2.7 Controller orchestration 624

14.2.8 Transaction boundaries 625

14.2.9 Encapsulating query shapes 626

14.2.10 Error mapping and domain-safe failures 627

14.2.11 Testing benefits 628

14.2.12 Summary 628

14.3 Error handling strategies 629

14.3.1 Goals of error handling . . . 629
14.3.2 Error categories . . . 630
14.3.3 Exception translation and domain error types . . . 631
14.3.4 Repository-level strategies . . . 632
14.3.5 Service and controller response mapping . . . 634
14.3.6 Transaction boundary and exception safety . . . 634
14.3.7 Retry policies for transient errors . . . 635
14.3.8 Logging and observability . . . 636
14.3.9 UI-safe error messages . . . 637
14.3.10 Mapping backend-specific errors . . . 637
14.3.11 Testing error handling . . . 637
14.3.12 Summary . . . 638

VII Business Logic Layer 639

15 Service Layer Design 640
15.1 Why a service layer matters . . . 640
15.1.1 The problem it solves . . . 640
15.1.2 What a service layer is (and is not) . . . 641
15.1.3 The key benefits in large systems . . . 642
15.1.4 Service layer shape in Modern C++ (pragmatic design) . . . 644
15.1.5 Wt-specific value: keeping widget code small and deterministic . . . 648
15.1.6 Common anti-patterns (and why they hurt) . . . 649
15.1.7 Practical checklist . . . 650
15.1.8 Outcome . . . 650
15.2 Isolating business rules . . . 650
15.2.1 What counts as a business rule? . . . 651

15.2.2 Why rules must be isolated 651
15.2.3 Architecture strategy: Functional Core, Imperative Shell 653
15.2.4 Where to place rules: a practical taxonomy 653
15.2.5 Service layer role: orchestration, not rule storage 658
15.2.6 Preventing rule duplication in the UI (Wt) 661
15.2.7 Techniques that strengthen rule isolation 663
15.2.8 Common anti-patterns (and how to avoid them) 664
15.2.9 Checklist for a clean isolation 665
15.2.10 Outcome 665
15.3 Improving testability 665
15.3.1 What we mean by "testability" in a service-layer design 666
15.3.2 How the service layer improves testability 666
15.3.3 Designing services for testability: concrete patterns 669
15.3.4 Example: a testable service use-case 671
15.3.5 Testing strategy: layering your tests correctly 675
15.3.6 Wt-specific guidance for testability 677
15.3.7 Common pitfalls that destroy testability 677
15.3.8 Practical checklist 678
15.3.9 Outcome 678

16 Controllers **679**
16.1 The role of controllers in Wt 679
16.1.1 What is a controller? 679
16.1.2 Why controllers matter in Wt 680
16.1.3 Controllers vs. Wt resources and widgets 681
16.1.4 Design principles for Wt controllers 682
16.1.5 Example: a REST controller for user registration 683
16.1.6 Routing and controllers 686

16.1.7 Controller responsibilities in detail 687
16.1.8 Testing controllers 688
16.1.9 Best practices for controllers in Wt applications 689
16.1.10 Outcome 689
16.2 Coordinating between UI and services 690
16.2.1 The core objective 690
16.2.2 Why this separation matters in Wt 691
16.2.3 Controller responsibilities when coordinating UI and services 692
16.2.4 A robust pattern: View → Controller → Service 694
16.2.5 Stable service result types enable clean coordination 694
16.2.6 Example: Registration flow coordinated by a controller 695
16.2.7 Handling dynamic UI updates without leaking business logic 699
16.2.8 Avoiding common coordination mistakes 701
16.2.9 A practical checklist for controller coordination 702
16.2.10 Outcome 702
16.3 Preventing logic leakage into the UI layer 703
16.3.1 What “logic leakage” looks like in Wt 703
16.3.2 Root causes in server-side UI frameworks 705
16.3.3 The boundary rule 705
16.3.4 The controller as the anti-leak barrier 706
16.3.5 A stable service API prevents UI decision-making 706
16.3.6 Bad vs. good: a leakage example 708
16.3.7 Key techniques to prevent leakage 711
16.3.8 Wt-specific leakage traps and how to avoid them 713
16.3.9 A practical checklist 714
16.3.10 Outcome 714

VIII Real-World Application Features 715

17 Advanced Forms 716

17.1 Professional validation techniques 716
17.1.1 Validation layers in a production-grade Wt application 717
17.1.2 Wt fundamentals: validators, visual feedback, and authoritative checks . 718
17.1.3 Technique 1: Per-widget validators for simple forms 719
17.1.4 Technique 2: Model-based validation with WFormModel and WTemplateFormView 720
17.1.5 Technique 3: Cross-field and conditional validation (real-world rules) . 725
17.1.6 Technique 4: Server-side uniqueness and anti-race validation 726
17.1.7 Technique 5: Validation timing (when to validate) 726
17.1.8 Technique 6: Canonicalization and safe bounds (security-grade validation) 727
17.1.9 Technique 7: Professional error reporting (messages and UX) 729
17.1.10 Technique 8: Testing validation professionally 729
17.1.11 Checklist: what "professional validation" means in Chapter 17 730
17.1.12 Outcome 731
17.2 Error messaging 731
17.2.1 Goals of professional error messaging 732
17.2.2 Error categories and their meaning 732
17.2.3 Field-level error messaging in Wt 734
17.2.4 Form-level error summaries 735
17.2.5 Stable service error categories 735
17.2.6 Controller-level error mapping 736
17.2.7 Security principles in error messaging 737
17.2.8 Internationalization-ready error messages 737
17.2.9 Visual and accessibility considerations 738
17.2.10 Testing error messaging 738

17.2.11 Checklist for professional error messaging . . . 738
17.2.12 Outcome . . . 739
17.3 Server-side UX considerations . . . 739
17.3.1 The server-side UX constraint model . . . 740
17.3.2 Principle 1: Optimize perceived responsiveness . . . 740
17.3.3 Principle 2: Prevent duplicate submissions and multi-click hazards . . . 741
17.3.4 Principle 3: Validate progressively, but confirm authoritatively . . . 742
17.3.5 Principle 4: Use clear error placement and recovery paths . . . 743
17.3.6 Principle 5: Keep long forms stable and navigable . . . 743
17.3.7 Principle 6: Design for session state and refresh safety . . . 744
17.3.8 Principle 7: Concurrency and conflict UX . . . 745
17.3.9 Principle 8: Accessibility and keyboard-first flow . . . 745
17.3.10 Principle 9: Data binding and safe re-rendering . . . 746
17.3.11 Principle 10: Professional messaging and status design . . . 747
17.3.12 A practical UX checklist for Wt advanced forms . . . 747
17.3.13 Outcome . . . 748

18 Data Tables and Views 749
18.1 Tables . . . 749
18.1.1 What makes tables a production feature? . . . 749
18.1.2 The Wt model/view approach . . . 750
18.1.3 Choosing the right table view approach . . . 750
18.1.4 Data table requirements: structure and responsibilities . . . 751
18.1.5 A professional table pipeline . . . 752
18.1.6 Defining a row DTO for tables . . . 752
18.1.7 Defining a paged result contract . . . 753
18.1.8 Table request parameters: sort, filter, paging . . . 753
18.1.9 Service interface for table data . . . 754

18.1.10 Implementing a Wt table model 754
18.1.11 Rendering a view with sorting and paging controls 756
18.1.12 Server-side sorting: safe and validated 758
18.1.13 Filtering and searching: predictable behavior 758
18.1.14 Pagination strategy: offset vs. keyset 759
18.1.15 Row actions: selection and commands 759
18.1.16 Rendering safety: escaping and output discipline 760
18.1.17 Performance and UX considerations 760
18.1.18 Testing strategy 761
18.1.19 Outcome 761
18.2 Pagination 761
18.2.1 What pagination must guarantee 762
18.2.2 Core pagination parameters 762
18.2.3 Page size discipline and safety 763
18.2.4 Stable ordering: the non-negotiable rule 764
18.2.5 Pagination strategies: offset vs. keyset 764
18.2.6 A unified service contract for paging 766
18.2.7 Computing total count: when to do it 766
18.2.8 Implementing pagination in a Wt table model 767
18.2.9 Pagination controls and UX behavior 768
18.2.10 Sorting and filtering interaction with pagination 769
18.2.11 Concurrency: what users see when data changes 769
18.2.12 Performance rules for production pagination 770
18.2.13 Professional checklist for Wt pagination 771
18.2.14 Outcome 771
18.3 Filtering 772
18.3.1 Filtering requirements in production-grade tables 772

18.3.2 Filtering is a contract: UI → controller → service → repository 773
18.3.3 Filter types you must support 773
18.3.4 Designing a filter request object (DTO) 775
18.3.5 Normalization and validation (controller/service level) 776
18.3.6 Filtering and authorization: never separate them 777
18.3.7 Repository query construction: safety and performance 777
18.3.8 Wt UI patterns for professional filtering 778
18.3.9 Filtering interaction with pagination and sorting 780
18.3.10 Empty states, partial matches, and user trust 780
18.3.11 Testing filtering professionally 781
18.3.12 Checklist for production-grade filtering 781
18.3.13 Outcome 782
18.4 Sorting 782
18.4.1 What sorting must guarantee 782
18.4.2 Sorting as an explicit contract 783
18.4.3 Server-side sorting is the default for production 783
18.4.4 Stable ordering: always add a tie-breaker 784
18.4.5 Sorting and security: whitelist, do not trust UI 784
18.4.6 Normalization rules for sorting 785
18.4.7 Sorting UX: predictable interactions 786
18.4.8 Sorting and performance engineering 786
18.4.9 Sorting in combination with filtering and pagination 787
18.4.10 Concurrency effects and user trust 788
18.4.11 Multi-column sorting (advanced) 788
18.4.12 Testing sorting professionally 789
18.4.13 Checklist for production-grade sorting 789
18.4.14 Outcome 790

19 File Uploads and Downloads **791**

19.1 File upload handling . . . 791

19.1.1 Server-side file handling principles . . . 791

19.1.2 Wt file upload controls . . . 792

19.1.3 Size limits and resource bounds . . . 793

19.1.4 Allowed types and extension validation . . . 793

19.1.5 Streaming uploads . . . 794

19.1.6 Integrating upload with services . . . 794

19.1.7 Safe storage strategies . . . 795

19.1.8 Virus/malware scanning . . . 795

19.1.9 Progress feedback and UX . . . 796

19.1.10 Error handling during upload . . . 796

19.1.11 Processing uploaded content . . . 797

19.1.12 Security boundary: access control for uploaded files . . . 797

19.1.13 Testing file upload handling . . . 798

19.1.14 Checklist for production-grade file uploads . . . 799

19.1.15 Outcome . . . 799

19.2 Security considerations . . . 800

19.2.1 Threat model for file features . . . 800

19.2.2 Core security principles . . . 801

19.2.3 Upload security controls . . . 802

19.2.4 Download security controls . . . 806

19.2.5 Cross-cutting concerns . . . 808

19.2.6 Common mistakes to avoid . . . 809

19.2.7 Testing security for file features . . . 809

19.2.8 Checklist: secure uploads and downloads . . . 810

19.2.9 Outcome . . . 811

19.3 Storage strategies 811
19.3.1 Core design goals for storage 812
19.3.2 A uniform storage abstraction 812
19.3.3 Storage option 1: Local filesystem storage 813
19.3.4 Storage option 2: Shared network storage (NFS/SMB or distributed FS) 815
19.3.5 Storage option 3: Database BLOB storage 816
19.3.6 Storage option 4: Object storage (S3-compatible) 817
19.3.7 Metadata storage and file identity 819
19.3.8 Integrity and immutability strategy 820
19.3.9 Retention, deletion, and compliance 820
19.3.10 Backup and disaster recovery considerations 821
19.3.11 A practical selection guide 821
19.3.12 Implementation checklist 822
19.3.13 Outcome 822

IX Performance, Security, and Stability 823

20 Performance in Wt Applications 824
20.1 Memory management 824
20.1.1 Understanding the Wt execution model 824
20.1.2 Rule 1: Prefer automatic storage and RAII 825
20.1.3 Rule 2: Minimize dynamic allocations in hot paths 825
20.1.4 Rule 3: Manage session and widget lifetimes explicitly 826
20.1.5 Rule 4: Use smart pointers for ownership clarity 826
20.1.6 Rule 5: Avoid global state that accumulates memory 827
20.1.7 Rule 6: Apply object pooling for frequent allocations 827
20.1.8 Rule 7: Use move semantics to avoid copies 828

20.1.9 Rule 8: Profile and measure memory usage 828
20.1.10 Rule 9: Prevent leaks across asynchronous tasks 829
20.1.11 Rule 10: Avoid accidental retention in closures 829
20.1.12 Rule 11: Bound caches with eviction policies 829
20.1.13 Rule 12: Watch fragmentation and defragmentation 830
20.1.14 Rule 13: Streaming large payloads instead of buffering 830
20.1.15 Rule 14: Guard against retain cycles 831
20.1.16 Rule 15: Monitor memory usage in production 831
20.1.17 Testing memory behavior 832
20.1.18 Checklist for memory-safe Wt applications 832
20.1.19 Outcome 833
20.2 Session scalability 833
20.2.1 What “session scalability” really means 834
20.2.2 The session cost model 834
20.2.3 Rule 1: Keep per-session state minimal and bounded 835
20.2.4 Rule 2: Prefer stateless services; avoid per-session heavy dependencies 836
20.2.5 Rule 3: Use session timeouts and cleanup hooks correctly 837
20.2.6 Rule 4: Avoid rebuilding large widget trees 838
20.2.7 Rule 5: Make “idle sessions” cheap 838
20.2.8 Rule 6: Control shared resource contention 839
20.2.9 Rule 7: Design for horizontal scaling 839
20.2.10 Rule 8: Use backpressure and bounded concurrency 840
20.2.11 Rule 9: Treat large tables as paged views, not session-owned datasets . 841
20.2.12 Rule 10: Measure session footprint and growth over time 842
20.2.13 Practical session scalability patterns in Wt 842
20.2.14 Checklist: session scalability readiness 844
20.2.15 Outcome 844

20.3 Threading model 844
20.3.1 Server architecture and concurrency overview 845
20.3.2 Thread safety and Wt application state 846
20.3.3 Event loop and reactor pattern 846
20.3.4 Request handling and worker thread pools 847
20.3.5 Posting work from non-UI threads 847
20.3.6 Avoiding blocking in event threads 848
20.3.7 Thread pool sizing and tuning 849
20.3.8 Multi-process and multi-node deployment 849
20.3.9 Common concurrency pitfalls and professional guidance 850
20.3.10 Outcome 850
20.4 Response optimization 851
20.4.1 What “response” means in Wt 851
20.4.2 Principle 1: Optimize the critical path (latency before throughput) . . . 852
20.4.3 Principle 2: Reduce response size by design 852
20.4.4 Principle 3: Exploit incremental UI updates (diff-based thinking) . . . 853
20.4.5 Principle 4: Cache and reuse what is stable 854
20.4.6 Principle 5: Use compression strategically 854
20.4.7 Principle 6: Minimize round-trips 855
20.4.8 Principle 7: Streaming for large responses 855
20.4.9 Principle 8: Optimize database interaction for response time 856
20.4.10 Principle 9: Control logging overhead in hot paths 856
20.4.11 Principle 10: Avoid accidental response bloat from UI state 857
20.4.12 Practical patterns for Wt response optimization 857
20.4.13 Measurement: optimization without metrics is guessing 858
20.4.14 Checklist: response optimization readiness 859
20.4.15 Outcome 860

21 Security in Wt Applications **861**

21.1 Session security 861

21.1.1 Threat model for sessions 861

21.1.2 Core principle: session security is layered 862

21.1.3 Transport security: TLS is mandatory 863

21.1.4 Session identifier properties 863

21.1.5 Cookie hardening 864

21.1.6 Session fixation prevention: rotate session ID on privilege change . . . 864

21.1.7 CSRF protection in a session-based system 865

21.1.8 Idle timeout and absolute lifetime 865

21.1.9 Logout semantics: invalidate server-side, not only client-side 866

21.1.10 Session binding and anomaly detection (defense-in-depth) 866

21.1.11 Cluster and multi-process considerations 867

21.1.12 Professional patterns in C++ for secure session primitives 867

21.1.13 Operational security: headers and policies 870

21.1.14 Auditing and monitoring 870

21.1.15 Checklist: secure sessions in Wt 871

21.1.16 Outcome 871

21.2 Password hashing 872

21.2.1 Why hashing; not encryption or plain storage 872

21.2.2 Professional algorithms: Argon2 and bcrypt 873

21.2.3 Salting and parameterization 873

21.2.4 Hash storage format 874

21.2.5 Verification flow 874

21.2.6 Parameter selection and cost tuning 874

21.2.7 Migration and algorithm agility 875

21.2.8 C++ integration patterns 875

21.2.9 Constant-time comparison . . . 877
21.2.10 Secure random for salt and secrets . . . 877
21.2.11 Protecting against offline attacks . . . 878
21.2.12 Avoiding anti-patterns . . . 878
21.2.13 Testing and validation . . . 878
21.2.14 Migration strategies . . . 879
21.2.15 Outcome . . . 879
21.3 HTTPS concepts . . . 880
21.3.1 HTTPS versus HTTP: what changes . . . 880
21.3.2 TLS building blocks . . . 881
21.3.3 HTTPS and Wt: typical deployment topologies . . . 883
21.3.4 Reverse proxy correctness: the “real scheme” problem . . . 884
21.3.5 HSTS: enforcing HTTPS at the browser . . . 884
21.3.6 Cookies and HTTPS: session security coupling . . . 885
21.3.7 TLS configuration goals: security and performance . . . 885
21.3.8 OCSP and revocation: practical view . . . 886
21.3.9 Certificate lifecycle and automation . . . 886
21.3.10 Mixed content, origin integrity, and application correctness . . . 887
21.3.11 WebSockets and HTTPS: WSS . . . 887
21.3.12 Practical edge configuration example (reverse proxy) . . . 887
21.3.13 Common mistakes to avoid . . . 889
21.3.14 Checklist: HTTPS readiness for Wt applications . . . 890
21.3.15 Outcome . . . 890
21.4 Critical security pitfalls . . . 890
21.4.1 Pitfall 1: Treating TLS as optional or inconsistent . . . 891
21.4.2 Pitfall 2: Weak session management (fixation, hijacking, long lifetimes) 891
21.4.3 Pitfall 3: Relying on “server-side UI” as an XSS shield . . . 892

21.4.4 Pitfall 4: Missing CSRF protection on state changes 893
21.4.5 Pitfall 5: Trusting client-provided metadata (MIME type, file name, IDs) 893
21.4.6 Pitfall 6: Insecure direct object references (IDOR) 894
21.4.7 Pitfall 7: SQL injection and unsafe query construction 895
21.4.8 Pitfall 8: Password storage mistakes 895
21.4.9 Pitfall 9: Unsafe file upload handling 896
21.4.10 Pitfall 10: Cross-thread UI access and race conditions 897
21.4.11 Pitfall 11: Logging secrets and sensitive data 897
21.4.12 Pitfall 12: Unbounded resource consumption (DoS by design) 898
21.4.13 Pitfall 13: Unsafe redirects and open redirect vulnerabilities 899
21.4.14 Pitfall 14: Incorrect trust boundary behind reverse proxies 899
21.4.15 Pitfall 15: “Security by UI” instead of server-side authorization 900
21.4.16 Pitfall 16: Missing secure headers and browser defenses 900
21.4.17 Pitfall 17: Insecure secrets management 901
21.4.18 Pitfall 18: Ignoring security testing and dependency risk 901
21.4.19 High-impact security checklist (production baseline) 902
21.4.20 Outcome 902

22 Logging and Debugging **904**

22.1 Logging architecture 904
22.1.1 Design goals 905
22.1.2 The three layers of logging 905
22.1.3 Log taxonomy: what to log 906
22.1.4 Structured logging: the default professional choice 908
22.1.5 Correlation and trace context 909
22.1.6 Severity levels and policy 910
22.1.7 Performance architecture: avoid logging as a bottleneck 910
22.1.8 Security architecture: redaction and minimization 913

22.1.9 Wt-specific logging integration points 914
22.1.10 Debugging architecture: logs + metrics + traces 914
22.1.11 Rotation, retention, and operational policy 915
22.1.12 A practical logging interface for a Wt application 915
22.1.13 Checklist: logging architecture readiness 916
22.1.14 Outcome 917
22.2 Error tracing 917
22.2.1 Goals of error tracing 917
22.2.2 Structured error context 918
22.2.3 Error categories 919
22.2.4 Capturing context at the throw site 919
22.2.5 Propagation and enrichment 920
22.2.6 Stack and location identification 920
22.2.7 Correlation identifiers 921
22.2.8 Integration with structured logs 921
22.2.9 Error tagging and marking expected vs unexpected 922
22.2.10 Error catch and report boundaries 922
22.2.11 Asynchronous and background work 923
22.2.12 Linking traces across dependent services 923
22.2.13 Semantic tagging of error fields 924
22.2.14 Error dashboards and alerting 924
22.2.15 Privacy and compliance in error traces 924
22.2.16 Debuggable failures without verbose logging 925
22.2.17 Testing error tracing 925
22.2.18 Checklist: error tracing readiness 926
22.2.19 Outcome 926
22.3 Production diagnostics 926

22.3.1 Diagnostics pillars: logs, metrics, traces, and profiles 927
22.3.2 Production-safe diagnostic design principles 928
22.3.3 Health endpoints and service readiness 929
22.3.4 Key production metrics for Wt applications 931
22.3.5 Structured error reporting and incident-grade context 933
22.3.6 Runtime debug switches (feature flags and diagnostic toggles) 934
22.3.7 Crash diagnostics: core dumps and post-mortem analysis 935
22.3.8 Memory diagnostics in production 936
22.3.9 Latency diagnostics: finding where time goes 937
22.3.10 Threading and concurrency diagnostics 938
22.3.11 Dependency diagnostics: database, storage, and external APIs 938
22.3.12 Safe production debugging workflow 939
22.3.13 Production diagnostics checklist 940
22.3.14 Outcome 941

X Deployment on Linux 942

23 Running Wt as a Server Application 943

23.1 Standalone server mode 943
23.1.1 When standalone mode is a good fit 943
23.1.2 Trade-offs versus FastCGI deployment 944
23.1.3 Core concept: the built-in HTTP(S) connector 945
23.1.4 The typical entry point: WRun and WServer 945
23.1.5 Configuration sources and precedence 947
23.1.6 Built-in server responsibilities and static resources 947
23.1.7 Threading and concurrency model in standalone mode 948
23.1.8 Ports, binding, and Linux networking posture 949

23.1.9 Reverse proxy fronting (recommended production layout) 949
23.1.10 systemd service integration 950
23.1.11 Graceful shutdown behavior 951
23.1.12 Deployment checklist for standalone mode 951
23.1.13 Outcome 952
23.2 Ports and networking basics 952
23.2.1 TCP and UDP basics 952
23.2.2 What is a port? 953
23.2.3 Binding addresses and interfaces 953
23.2.4 Reverse proxying and port redirection 954
23.2.5 Firewalls and access control 955
23.2.6 Port ranges and ephemeral ports 955
23.2.7 Hostnames, DNS, and virtual hosts 956
23.2.8 Health check endpoints and monitoring ports 956
23.2.9 TLS and port assignment 957
23.2.10 IPv4 and IPv6 considerations 957
23.2.11 Session affinity and load balancing 958
23.2.12 Port exhaustion and connection limits 958
23.2.13 Practical network configuration for Wt with reverse proxy 959
23.2.14 Checklist: networking readiness for Wt on Linux 959
23.2.15 Outcome 960

24 Nginx and Reverse Proxy Setup **961**
24.1 Nginx configuration 961
24.1.1 Recommended topology 962
24.1.2 Core reverse-proxy rules for Wt 962
24.1.3 A production baseline configuration (HTTP redirect + HTTPS reverse proxy) 963

24.1.4 Important details and why they matter 966
24.1.5 Serving static files efficiently (recommended) 967
24.1.6 Rate limiting and request hardening 967
24.1.7 Uploads and downloads 968
24.1.8 Correct client IP handling behind proxies 969
24.1.9 Scaling out: multiple Wt instances and session affinity 969
24.1.10 Operational hygiene: logs, reloads, and validation 970
24.1.11 A concise production checklist 971
24.1.12 Outcome 971
24.2 WebSocket forwarding 971
24.2.1 WebSocket upgrade: the essential mechanism 972
24.2.2 Production baseline: Nginx forwarding to Wt 973
24.2.3 Why each directive matters 975
24.2.4 Forwarding on a dedicated WebSocket path 976
24.2.5 Client IP and scheme correctness 977
24.2.6 Load balancing and session affinity 978
24.2.7 Common failure patterns and how to prevent them 978
24.2.8 Production checklist for WebSocket forwarding 980
24.2.9 Outcome 981
24.3 HTTPS integration 981
24.3.1 Deployment model: TLS terminates at Nginx 981
24.3.2 Certificate lifecycle: obtain, renew, reload 982
24.3.3 Baseline HTTPS configuration: redirect + TLS + proxy 983
24.3.4 Why each HTTPS component matters 986
24.3.5 Cookie and session security coupling with HTTPS 987
24.3.6 Security headers and HTTPS 987
24.3.7 WebSockets over HTTPS: WSS 988

24.3.8 Uploads, buffering, and HTTPS . . . 989
24.3.9 Correct client IP behavior behind TLS termination . . . 989
24.3.10 Operational hardening checklist for HTTPS . . . 989
24.3.11 Outcome . . . 990

25 systemd and Professional Service Management 991
25.1 Creating a systemd service . . . 991
25.1.1 Service unit structure . . . 992
25.1.2 Example service unit for a Wt application . . . 992
25.1.3 Key directives explained . . . 993
25.1.4 Creating a dedicated service user . . . 994
25.1.5 Enabling and managing the service . . . 995
25.1.6 Handling configuration changes . . . 995
25.1.7 Resource limits and service tuning . . . 996
25.1.8 Logging and observability . . . 996
25.1.9 Graceful shutdown and signals . . . 997
25.1.10 Security hardening considerations . . . 997
25.1.11 High-availability considerations . . . 998
25.1.12 Monitoring systemd services . . . 998
25.1.13 Outcome . . . 999
25.2 Automatic restarts . . . 999
25.2.1 What systemd considers a failure . . . 1000
25.2.2 Core restart directives . . . 1001
25.2.3 A professional baseline restart configuration . . . 1002
25.2.4 Rate limiting: preventing restart storms . . . 1003
25.2.5 Distinguishing expected exits from failures . . . 1004
25.2.6 Choosing the correct `Type=` and its impact on restarts . . . 1004
25.2.7 Handling OOM kills and memory-related restarts . . . 1005

25.2.8 Restart behavior and dependencies 1005
25.2.9 Observability of restarts 1006
25.2.10 Recommended restart policy for Wt servers 1007
25.2.11 Checklist: automatic restarts done professionally 1008
25.2.12 Outcome 1009
25.3 Basic monitoring 1009
25.3.1 Monitoring goals 1009
25.3.2 Systemd service health 1010
25.3.3 Resource utilization monitoring 1010
25.3.4 Basic health endpoints 1012
25.3.5 Latency and error monitoring 1012
25.3.6 Dependency availability checks 1013
25.3.7 Logs as a monitoring source 1014
25.3.8 Example: simple Nginx access log analysis 1014
25.3.9 Alerting on thresholds 1015
25.3.10 Monitoring tooling options 1015
25.3.11 Metrics collection and exporters 1016
25.3.12 Health checks integrated with orchestration 1016
25.3.13 Resource limit proximity monitoring 1016
25.3.14 Combining metrics and logs for richer signals 1017
25.3.15 Session-level monitoring signals 1017
25.3.16 Alert automation and escalation policies 1017
25.3.17 Documentation and runbooks 1018
25.3.18 Outcome 1018

26 Mastery and Scalability **1019**

27 Configuration Management **1020**

27.1 Configuration files vs environment variables 1020

27.1.1 Definitions and scope 1021

27.1.2 Why this choice matters in scalable systems 1021

27.1.3 Configuration files: strengths and weaknesses 1022

27.1.4 Environment variables: strengths and weaknesses 1024

27.1.5 A professional position: use both, with explicit roles 1025

27.1.6 Recommended precedence model 1026

27.1.7 Validation: the non-negotiable requirement 1026

27.1.8 Secrets: treat them differently 1027

27.1.9 systemd-friendly patterns 1027

27.1.10 Designing the configuration surface for Wt applications 1028

27.1.11 C++ implementation pattern: typed config + layered sources 1029

27.1.12 Operational patterns for scalability 1032

27.1.13 Decision guide 1033

27.1.14 Checklist: professional configuration management 1034

27.1.15 Outcome 1034

27.2 Managing production secrets 1035

27.2.1 Core principles of secret management 1035

27.2.2 Acceptable secret storage mechanisms 1036

27.2.3 Runtime secret retrieval 1038

27.2.4 In-memory secret handling 1039

27.2.5 Secret rotation strategy 1039

27.2.6 Secrets and configuration files 1040

27.2.7 systemd integration 1041

27.2.8 Audit and operational controls 1041

27.2.9 Common anti-patterns 1042

27.2.10 Checklist: production-ready secret management 1042
27.2.11 Outcome 1043

28 Scalability and Maintainability 1044
28.1 Scaling concepts 1044
28.1.1 1) What does it mean to scale? 1045
28.1.2 2) Vertical scaling vs horizontal scaling 1045
28.1.3 3) The state problem: stateless vs stateful scaling 1047
28.1.4 4) Scaling the connection model 1048
28.1.5 5) Bottleneck taxonomy: where systems usually fail first 1048
28.1.6 6) Caching as a scaling tool 1051
28.1.7 7) Load balancing and session affinity 1051
28.1.8 8) Decomposition: scaling the architecture, not just the process 1052
28.1.9 9) Concurrency model and safe scaling in Modern C++ 1053
28.1.10 10) Backpressure and graceful degradation 1054
28.1.11 11) Maintainability as a scaling multiplier 1055
28.1.12 12) Operational scaling: release and failure management 1056
28.1.13 13) Practical scaling model for Wt deployments 1056
28.1.14 14) Scaling checklist: concepts that must be explicit 1057
28.1.15 Outcome 1058
28.2 When load balancing becomes necessary 1058
28.2.1 1) What load balancing actually solves 1059
28.2.2 2) The objective triggers: measurable signals 1060
28.2.3 3) The architecture triggers: structural reasons 1062
28.2.4 4) The operational triggers: when the team needs it 1063
28.2.5 5) The Wt-specific reality: affinity and long-lived connections 1064
28.2.6 6) Practical thresholds: when it becomes unavoidable 1065
28.2.7 7) Introducing load balancing safely 1066

28.2.8 8) Minimal conceptual Nginx load balancing example 1067
28.2.9 9) Checklist: deciding to load balance 1068
28.2.10 Outcome 1068
28.3 Realistic limits of Wt 1069
28.3.1 1) The fundamental constraint: server-side state 1069
28.3.2 2) Memory footprint scales with sessions 1070
28.3.3 3) Concurrency and connection scaling limits 1071
28.3.4 4) Sticky sessions are often required 1072
28.3.5 5) Failover behavior is more complex than stateless services 1072
28.3.6 6) Server-side rendering and update cost 1073
28.3.7 7) Long-running tasks can damage interactive responsiveness 1074
28.3.8 8) Observability and debugging limits at scale 1074
28.3.9 9) Client-side ecosystem and UI constraints 1075
28.3.10 10) Realistic scalability envelope: what Wt can and cannot do well . . . 1075
28.3.11 11) Practical strategies to push limits safely 1076
28.3.12 12) A concrete capacity mindset for Wt deployments 1078
28.3.13 Outcome 1078

Conclusion **1081**

29 When to Use Wt and When Not To **1081**

When to Use Wt and When Not To **1081**
Comprehensive review 1081
Realistic comparison with Node.js and Spring 1091
Practical guidance for teams and projects 1104

Appendices **1115**

Appendix A Common Errors and Troubleshooting 1115

Appendix B Ready-to-Use CMake Templates 1130

Appendix C systemd and Nginx Configuration Templates 1145

Appendix D Widget Cheat Sheet 1153

Appendix E Roadmap for Large-Scale Wt Projects 1167

References **1179**

Author's Introduction

Modern software engineering has gradually accepted a common assumption: that "web development" must be dominated by client-heavy stacks and dynamic languages, while C++ remains confined to systems software, embedded environments, and performance-critical backends hidden behind layers of services. This assumption is not a law of nature. It is a reflection of mainstream tooling, market momentum, and the convenience of rapid prototyping. It is not a proof of architectural superiority.

This book was written to demonstrate a different engineering reality: it is possible to build serious, maintainable, production-grade web applications in C++ without surrendering the discipline that makes C++ valuable. When the problem domain requires strong correctness guarantees, predictable performance, long-lived state, controlled resource usage, strict encapsulation, and precise operational behavior, server-side C++ can be not only viable but strategically superior.

Wt Toolkit represents a deliberate design direction that is frequently misunderstood. It does not aim to imitate a JavaScript ecosystem. It is not a thin wrapper around HTML templating, nor a short-lived request handler intended to be restarted and scaled without state. Instead, Wt enables a server-side application model in which the web UI is expressed through C++ types, event handling is explicit, state is maintained intentionally, and the application behaves like a long-running system. In other words, Wt treats a web application as a real program with structure, invariants, and lifetimes, rather than a scattered set of scripts and templates.

The primary goal of this book is to provide a complete and technically rigorous pathway for

mastering Wt in a way that matches professional expectations. This is not a collection of disconnected snippets, nor a superficial overview of widgets. The reader is guided from first principles to production deployment by building a realistic application from the ground up. The project is not an educational toy; it is a structured system designed to teach architecture, separation of concerns, data access, authentication, configuration management, logging, performance, security discipline, and operational deployment on Linux.

The book is written for developers who already understand programming and C++ fundamentals, and who want to apply that knowledge to the web without abandoning engineering standards. If you have built non-trivial C++ software before, you already possess the mindset required for Wt; the main adjustment is conceptual: to treat the web as an event-driven, stateful interaction layer over a long-running server application. If you come from stateless web frameworks, you will need to re-evaluate how state, sessions, concurrency, and resource ownership are expressed and controlled. This book is structured to build that mental model carefully and systematically.

A central theme throughout the chapters is professional responsibility. In production systems, correctness is not a matter of passing local tests. It includes operational behavior, error handling, safe defaults, observability, and realistic threat models. It includes the ability to reason about failure modes, concurrency boundaries, and resource limits. It includes designing interfaces that remain stable under change, and building systems that can be maintained by teams over long lifecycles. These values are native to serious C++ engineering, and Wt allows them to be applied directly to web applications.

For that reason, the code in this book follows a strict set of practical principles:

- The UI layer must remain free of business rules and database logic.
- Controllers coordinate interaction; services own domain rules; models remain portable C++ types.
- Every feature is introduced with a reason, implemented in the live project, and framed

with common pitfalls.

- Deployment is treated as part of the engineering process, not an afterthought.
- The application is designed to remain stable under growth: more pages, more data, more users, and more requirements.

This is also a teaching book. If you plan to train others, you need more than code. You need an explanation strategy that builds competence without hiding complexity. Each major concept is approached in a repeatable pattern: first the purpose, then the mechanism, then correct usage, then failure cases, and finally professional rules of thumb. This approach is suitable for self-study, workshops, and structured courses.

The deployment aspect is treated with equal seriousness. Many "web framework" books stop at local execution, leaving production behavior vague. Here, we complete the engineering loop: configuration design, reverse proxy integration, HTTPS considerations, service management using `systemd`, and stable operational logging. The goal is for the reader to finish with a system that can be hosted responsibly on Linux with predictable behavior.

The web will continue to evolve. Tools will come and go. Yet core engineering requirements remain stable: correctness, maintainability, security, performance, and operational clarity. C++ remains one of the few languages that can satisfy these requirements when used with discipline. Wt offers a practical path to bring those strengths into the web domain without forcing the developer to abandon the language, abstractions, and rigor they trust.

To keep the book concrete, we anchor the learning journey in a single cohesive project. The intent is not to create a "demo" but to cultivate professional skill: the ability to design, implement, and operate a server-side C++ web application as a complete system.

If you read this book carefully, implement the project step by step, and reflect on the architectural decisions, you will gain more than familiarity with Wt. You will gain a transferable engineering capability: building web applications as disciplined C++ systems, designed and deployed for real production constraints.

A short note on code style and toolchains

The examples are written with modern C++ practices in mind and are structured to be readable, testable, and maintainable. The build system is based on CMake, and the deployment chapters assume a Linux server environment with a reverse proxy. Where appropriate, the book highlights portability considerations and outlines alternatives.

A minimal conceptual sketch

The following snippet is intentionally minimal. It exists only to highlight the fundamental model: a long-running server process that constructs a `WApplication` instance per session and renders the UI through C++.

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WEnvironment.h>
#include <Wt/WServer.h>

class App final : public Wt::WApplication {
public:
  explicit App(const Wt::WEnvironment& env) : Wt::WApplication(env) {
    setTitle("Modern C++ Web Development with Wt");
    root()->addWidget(std::make_unique<Wt::WContainerWidget>());
  }
};

int main(int argc, char** argv) {
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);
  server.addEntryPoint(Wt::EntryPointType::Application,
```

```
    [](const Wt::WEnvironment& env) { return std::make_unique<App>(env); });
  return server.run();
}
```

This book will turn this small idea into a complete production-grade system, built with a professional architecture, real features, and a deployment model suitable for serious environments.

Stay Connected

For more discussions and valuable content about **Modern C++ Web Development with Wt Toolkit**, I invite you to follow me on **LinkedIn**:

`https://linkedin.com/in/aymanalheraki`

You can also visit my personal website:

`https://simplifycpp.org`

`Ayman Alheraki`

Book Preface

This book is a practical and engineering-focused guide to building server-side web applications in modern C++ using the Wt Toolkit. It was written to address a recurring gap in the C++ ecosystem: while C++ is widely trusted for correctness, performance, and long-lived systems, developers who want to apply the same discipline to web applications often find that most web literature assumes a different world—one dominated by client-heavy stacks, short-lived processes, and rapid-change tooling.

Wt is intentionally different. It models web applications as long-running server programs, where state is explicit, lifetimes matter, and behavior can be reasoned about using the same principles that guide professional C++ development. This book treats that model as a strength, not an anomaly. The intent is to show how to design, implement, and deploy Wt applications with the rigor expected in serious production environments.

What this book is

This book is both a learning path and a production handbook. It is designed around three complementary goals:

- **Teach the Wt mental model clearly.** The reader will understand how Wt applications behave as processes, how sessions map to application instances, and how event-driven interaction is expressed and managed.

- **Build a complete real-world application.** The book follows a single live project that evolves chapter by chapter into a deployable system with authentication, persistence, configuration, logging, and operational structure.
- **Develop transferable engineering habits.** The architecture, layering, and operational practices demonstrated here generalize beyond Wt and remain valuable across server-side C++ systems.

The book is written in an academic-professional tone. It emphasizes correctness, maintainability, and realism. Each major topic is introduced because it solves a concrete problem in the project, and it is accompanied by design rationale, common pitfalls, and practical rules of thumb.

What this book is not

To keep the book aligned with production practice, it avoids several patterns that frequently produce fragile systems:

- It is not a collection of disconnected widget examples.
- It does not rely on framework "magic" that hides state, ownership, or control flow.
- It does not treat deployment as an optional afterthought.
- It does not optimize for superficial brevity at the expense of correct design.

The goal is to build systems that can survive requirement growth, team handoffs, security constraints, and operational stress.

Audience and prerequisites

This book assumes you already understand core C++ concepts, including object lifetimes, RAII, standard containers, error handling strategies, and basic concurrency vocabulary. Prior experience with web development is helpful but not required. However, the book is intentionally written for readers who want more than a quick introduction. It is aimed at developers who want to build and operate web systems professionally.
If you are new to CMake, Linux service management, or reverse proxy configuration, you will still be able to follow the book. Those topics are introduced gradually and applied only as needed for the project. The deployment chapters focus on the essential, stable techniques required to host Wt responsibly.

The learning method used in this book

The book follows a consistent teaching pattern for most features:

1. **Motivation and constraints.** What problem are we solving, and what production constraints matter?
2. **Design choice.** Which architectural decision supports maintainability and correctness?
3. **Implementation.** The feature is added to the live project in executable code.
4. **Failure modes and pitfalls.** What can go wrong, and how do we prevent it?
5. **Operational implications.** How does this feature behave under load, during failure, and in deployment?

This method is deliberate. Wt is most valuable when it is used with discipline. A reader should be able to explain not only *how* a feature was implemented, but *why* it is implemented that way, and what guarantees it provides.

The live project: scope and rationale

A production-grade framework cannot be learned fully through isolated samples. Therefore, the book is built around a single cohesive application that exercises the essential capabilities of Wt. The project is chosen to be realistic and broadly relevant: a server-side web system with authenticated users, a structured UI layout, persistent data, and operational deployment requirements.
Throughout the book, the project will demonstrate:

- A clean separation of responsibilities: UI, controllers, services, models, and database access.
- Authentication and session management with safe defaults and explicit boundaries.
- A data layer that supports evolution and avoids accidental coupling.
- Observability via logging and clear error handling paths.
- Deployment on Linux with reverse proxy integration and service management.

The goal is for readers to finish with a system that can be hosted, monitored, and maintained—not just executed locally.

Architectural rules enforced by the book

To ensure the project remains professional as it grows, the book follows a small set of strict architectural rules:

- **No database logic in UI widgets.** Widgets render and interact; persistence belongs elsewhere.

- **No business rules in UI.** Domain rules belong in services, not event handlers.
- **Explicit session and user context.** Session state is treated as an owned resource with clear lifetime.
- **Configurable behavior.** Production parameters are not hard-coded; configuration is designed as a first-class concern.
- **Operational readiness.** Logging, error handling, and deployment are part of the engineering definition of "done".

These rules are not theoretical. They are derived from common failure patterns in real systems: tight coupling, untestable logic, hidden state, and fragile deployments.

A note on security, correctness, and "most recent" practice

In production web systems, security and correctness are not optional features. They are core properties that must be intentionally designed and verified. This book emphasizes stable engineering practices rather than fashionable trends. Where modern practice matters—for example, safe password storage, secure session handling, and HTTPS deployment—the book adopts approaches that are widely accepted in professional environments and supported by well-established guidance.

How to use this book

You can use this book in two effective ways:

- **Project-first reading.** Follow the chapters sequentially, implement the live project step by step, and keep each stage runnable.

- **Reference-first reading.** Use the architecture and deployment chapters as a production handbook once you already have a basic Wt application.

If you intend to teach Wt to others, treat each chapter as a self-contained module with a defined outcome. Encourage learners to run the system after every chapter and to observe behavior under intentional errors. The fastest path to competence is not memorization; it is repeated, correct execution under realistic constraints.

A minimal build-and-run checkpoint

The book assumes a CMake-based workflow. The exact structure will be introduced early and refined throughout. The following snippet is included as a conceptual checkpoint: a professional workflow requires reproducible builds and explicit dependencies, even when the application is small.

```
cmake_minimum_required(VERSION 3.20)
project(wt_app LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Wt REQUIRED COMPONENTS Wt HTTP)

add_executable(wt_app main.cpp)
target_link_libraries(wt_app PRIVATE Wt::Wt Wt::HTTP)
```

From this foundation, the book will evolve the build system, project layout, configuration, and deployment model into a complete production-ready structure.

Part I

Conceptual and Technical Foundations

Why Wt? Why C++ on the Web?

1.1 Why Wt was created

1.1.1 The engineering gap Wt addresses

Wt was created to address a persistent engineering gap between two worlds that, in practice, often overlap:

- **Systems-grade engineering**, where software is built with explicit lifetimes, predictable performance, strong typing, and disciplined architecture.
- **Mainstream web development**, where development speed and client-side ecosystems often dominate, and where application behavior may become distributed across multiple languages, runtimes, and moving parts.

In many organizations, this gap becomes visible when web applications evolve beyond simple pages into long-lived systems with strict requirements. At that point, teams frequently rediscover the same constraints that systems programmers have always had to manage: controlled resource usage, correctness under concurrency, strict security boundaries, stable deployment, and maintainability over years rather than months.
Wt exists to make it practical to build web applications while keeping the core engineering advantages of C++: explicit ownership, deterministic destruction (RAII), static type

guarantees, and performance transparency. The framework was designed for cases where a web application should be treated as a *real program* running as a long-lived process, not merely as templates plus short-lived request handlers.

1.1.2 The central idea: server-side stateful interaction

A defining motivation behind Wt is that many web applications are inherently stateful. Users authenticate, navigate through workflows, filter and manipulate datasets, and interact with UI components that have natural lifetimes. Traditional stateless request/response frameworks can support such applications, but they typically require a significant amount of additional state management infrastructure, and they often push complexity either into the client or into layered service architectures.
Wt takes a different stance: it models interactive web applications in a way that resembles event-driven GUI systems, while still running on a web server. The core idea can be expressed as follows:

- The server maintains application state explicitly.
- The browser acts as a rendering and interaction endpoint.
- User interactions are handled as events that update server-side state and UI.

This approach reduces accidental complexity for certain classes of applications, because state and behavior can remain in one place: a strongly typed C++ codebase.

1.1.3 A deliberate alternative to client-heavy stacks

Wt was created not to compete with JavaScript frameworks on their own terms, but to offer a deliberate alternative with different engineering tradeoffs. In many modern stacks, the UI and business logic are split across multiple layers:

- client-side UI framework and tooling,
- server-side APIs and serialization formats,
- authentication and session strategies,
- separate build pipelines,
- cross-language debugging and performance profiling.

This architecture can be highly effective, but it also increases the number of moving parts. Wt was designed for situations where a team prefers to keep the application model, UI logic, and domain rules in a single language and runtime, reducing fragmentation and preserving C++-level control.
Wt does not eliminate the web; it changes the center of gravity. The browser remains the client, but the application model remains primarily server-side, expressed with C++ abstractions.

1.1.4 Type safety and explicitness as first-class goals

A major reason Wt exists is that C++ provides a level of explicitness and control that many production systems rely on:

- **Strong static typing** for UI states, domain models, and invariants.
- **RAII** for deterministic resource management (memory, file handles, database transactions, locks).
- **Performance predictability** by avoiding hidden runtime layers where possible.
- **Controlled abstraction** that can be reasoned about and audited.

Wt leverages these characteristics by allowing UI composition and event handling to be expressed directly in C++ types, rather than distributing UI state across dynamic runtime boundaries.

1.1.5 The UI-as-widgets model and event-driven programming

Wt adopts a widget-based model for building web user interfaces. This choice is not cosmetic; it reflects a design motivation: interactive user interfaces are naturally composed of components with lifetimes, state, and event handlers. A widget model provides a structured way to represent that reality.
In practice, this means:

- UI is built by composing widgets into containers and layouts.
- User actions are handled through events (signals) that invoke C++ code.
- State can live in well-defined C++ objects, with explicit ownership and lifetime.

The following snippet illustrates the conceptual style: UI composition and event wiring in C++, using a typed event handler.

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
#include <Wt/WEnvironment.h>

class App final : public Wt::WApplication {
public:
  explicit App(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Wt: Stateful C++ Web Application");

    auto status = root()->addWidget(std::make_unique<Wt::WText>("Ready."));
```

```
    auto btn    =
    ↪  root()->addWidget(std::make_unique<Wt::WPushButton>("Run action"));

    btn->clicked().connect([status] {
      status->setText("Action handled on the server.");
    });
  }
};
```

This model is one of the reasons Wt was created: it enables a disciplined, component-based approach to interactive web UI without forcing a split between languages for UI behavior.

1.1.6 Operational clarity: the application as a long-running service

Another motivation behind Wt is operational clarity. Many production systems benefit when the application behaves like a managed service:

- predictable startup and shutdown,
- explicit configuration,
- structured logging,
- well-defined resource ownership,
- service supervision (for example via process managers on Linux).

Wt aligns with this by encouraging a model where the web application is deployed as a server program. The application is not merely a set of scripts executed per request; it is a process with managed state and defined lifetimes.

1.1.7 Where Wt is most justified

Wt was created for use cases where the benefits of server-side C++ and stateful interaction outweigh the costs of operating a compiled service. Typical justified contexts include:

- internal enterprise applications and administrative consoles,
- systems with complex workflows and rich server-side validation,
- applications requiring strict performance predictability,
- environments where deployment stability and long-term maintainability dominate,
- teams already invested in C++ codebases, tooling, and engineering standards.

1.1.8 Design tradeoffs and the boundaries of the approach

Wt is not a universal replacement for all web approaches. Its creation reflects a specific engineering choice, and that choice has tradeoffs:

- A compiled, long-running process requires careful deployment and supervision.
- Stateful sessions must be managed with discipline to remain scalable.
- Client-heavy experiences may still be better served by dedicated frontend stacks.

These tradeoffs do not invalidate Wt; they define its intended scope. Wt was created so that, when the application is best modeled as a structured, stateful, long-lived C++ system, the web becomes a natural and practical delivery channel rather than a forced compromise.

1.1.9 Summary

Wt was created to make a particular class of web applications **engineerable** in the C++ sense: strongly structured, explicit, maintainable, and operationally clear. Its motivation is not novelty, but the belief that the web can host serious systems without requiring teams to abandon the discipline of systems programming. The framework exists to bring the strengths of modern C++ to interactive server-side web applications through a stateful, event-driven, widget-based model designed for production realities.

1.2 The fundamental difference between Wt and JavaScript frameworks

1.2.1 A difference of architecture, not syntax

The fundamental difference between Wt and most modern JavaScript frameworks is not the user interface technology in the browser, but the *location of the application model* and the *distribution of responsibility* across the system.

In a typical JavaScript framework ecosystem, the dominant assumption is that the browser is the primary runtime for the user interface and a substantial portion of application behavior. The server is frequently treated as an API provider, responsible for authentication, data access, and business operations, while the client manages rendering, interaction, and often significant state transitions.

Wt adopts a different model. It treats the application as a long-running server program written in C++, where the application state, interaction logic, and UI composition live primarily on the server. The browser becomes an interaction endpoint that renders a representation of server-side widgets and sends user events back to the server.

This yields two distinct paradigms:

- **Client-driven UI model (common JavaScript approach):** the browser hosts most UI state and behavior; the server provides APIs and data.
- **Server-driven UI model (Wt approach):** the server hosts UI state and behavior; the browser renders and forwards events.

Both models can build successful systems. The difference is which side is treated as the authoritative source of UI state and behavior, and how complexity is partitioned.

1.2.2 Where the "truth" lives: state ownership and authority

A practical way to understand the difference is to ask a simple question:

> Where is the authoritative state of the user interface stored and evolved?

In client-heavy JavaScript frameworks, the authoritative UI state is typically in the browser: component state, routing state, view state, cached API results, client-side validation state, and often optimistic updates. The server may also maintain state (sessions, tokens, database state), but the immediate UI truth is often client-owned.
In Wt, the authoritative UI state is server-owned. The widget tree, the current view, and the interaction state live in C++ objects with explicit lifetimes. The browser reflects this state. User events are forwarded to the server, which updates the widget state and propagates the necessary changes back to the browser.
This difference influences:

- correctness and invariants (where they are enforced),
- security boundaries (where sensitive logic resides),
- debugging and observability (what you can inspect and log),
- performance tradeoffs (network chatter vs client computation),
- deployment and scaling strategies (stateful vs stateless assumptions).

1.2.3 Programming model: event-driven server objects vs client components

JavaScript frameworks commonly expose a component model where UI is expressed as a function of state, and the framework updates the DOM based on diffing and reconciliation. The programming style typically revolves around:

- reactive state updates,
- client-side rendering,
- asynchronous calls to APIs,
- bundlers and client toolchains,
- browser runtime constraints.

Wt exposes a widget model expressed in C++. UI is composed by instantiating widgets and containers, and behavior is attached via server-side event handlers. Instead of managing a DOM tree directly, you manage a widget tree. Instead of client-side callbacks that mutate browser state, you write server-side handlers that mutate C++ state.
The difference is best seen in a conceptual comparison: a button click is processed where the application model lives.

JavaScript framework style (conceptual)

```
// Conceptual example: client-owned UI state
function Counter() {
  const [n, setN] = useState(0);

  return (
    <button onClick={() => setN(n + 1)}>
```

```
      Count: {n}
    </button>
  );
}
```

Wt style (conceptual)

```
// Conceptual example: server-owned UI state
#include <Wt/WContainerWidget.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class CounterView final : public Wt::WContainerWidget {
public:
  CounterView() {
    label_ = addWidget(std::make_unique<Wt::WText>("Count: 0"));
    btn_   = addWidget(std::make_unique<Wt::WPushButton>("Increment"));

    btn_->clicked().connect([this] {
      ++n_;
      label_->setText("Count: " + std::to_string(n_));
    });
  }

private:
  int n_{0};
  Wt::WText* label_{nullptr};
  Wt::WPushButton* btn_{nullptr};
};
```

The essential point is not the syntax, but the runtime authority: in the JavaScript style, state is updated in the browser; in Wt, state is updated in the server process.

1.2.4 The toolchain boundary: one runtime vs two runtimes

Most JavaScript framework architectures implicitly create a two-runtime system:

- the browser runtime (UI logic, rendering, client state),
- the server runtime (APIs, persistence, authorization, business logic).

This is not inherently negative; it is a deliberate design that enables rich client experiences and offline-friendly behavior. However, it also introduces common engineering costs:

- duplicated validation rules (client and server),
- distributed debugging (browser devtools plus server logs),
- serialization boundaries (JSON payloads, schema drift),
- more moving parts in deployment pipelines,
- increased risk of inconsistent behavior between layers.

Wt reduces the severity of these costs for many applications because the authoritative UI behavior is server-side. You still have a browser as a client, but you do not necessarily need to replicate large parts of domain logic across two runtimes. The application logic, UI logic, and domain rules can remain within a single C++ codebase, which can improve consistency and auditability.

1.2.5 Statefulness, sessions, and scalability assumptions

Modern web stacks often optimize for statelessness at the server boundary. Stateless APIs scale horizontally with relative simplicity, and client-owned state supports resilience and caching strategies. Many JavaScript-oriented architectures therefore assume that server-side components should be replaceable and restartable without losing UI state.
Wt assumes statefulness as a first-class property. Each user session has associated server-side state, including widget state and any session-bound domain context. This makes certain workflows simpler and more robust from a correctness perspective, but it changes scaling assumptions. When scaling Wt systems, one must plan around session state management, memory usage, and server-side concurrency.
This difference is fundamental:

- JavaScript frameworks commonly pair with stateless server APIs by default.
- Wt commonly pairs with stateful server sessions by design.

Neither is universally superior. The correct choice depends on the system's goals: interactive workflows, security requirements, data size, concurrency patterns, and operational constraints.

1.2.6 Security boundaries and trust distribution

In client-heavy architectures, the client is untrusted by default, and sensitive rules must always be revalidated on the server. While this is correct security practice, it often leads to duplicated logic and careful synchronization between client convenience and server authority.
Wt centralizes more behavior on the server. This does not eliminate the need for security discipline, but it shifts the natural trust boundary: UI decisions, workflow transitions, and sensitive computations can remain server-side, reducing the temptation to treat client-side behavior as authoritative. In practice, this can simplify the enforcement of invariants, because the server owns both the UI state and the domain logic.

1.2.7 User experience: responsiveness vs round-trips

JavaScript frameworks often deliver maximum responsiveness by executing UI logic locally in the browser, avoiding network round-trips for many interactions. Wt can also provide responsive experiences, but its default interaction model frequently involves server round-trips for event handling and UI updates. The framework uses efficient update mechanisms, but the interaction design must still be mindful of latency.
This introduces a classic engineering tradeoff:

- client-heavy frameworks trade complexity and duplication for local responsiveness,
- server-driven frameworks trade some round-trips for centralized correctness and simpler distribution of logic.

Well-designed Wt applications minimize unnecessary round-trips by batching updates, designing interaction flows carefully, and avoiding overly chatty UI behaviors.

1.2.8 Maintainability: consistency and architectural gravity

One of the most practical differences in professional environments is architectural gravity: systems tend to grow toward the model that makes change safer.
In JavaScript-centric stacks, large projects often introduce patterns, shared schemas, code generation, and duplicated validation frameworks to keep client and server consistent. In Wt systems, the center of gravity often remains in the C++ codebase: domain logic, UI state, and workflow control are naturally unified.
For organizations with strong C++ expertise, this can be a strategic advantage:

- fewer cross-language boundary failures,
- consistent typing and invariants,
- unified profiling and logging culture,

- easier integration with existing C++ libraries and systems.

1.2.9 Summary

The fundamental difference between Wt and JavaScript frameworks is a difference of **application ownership and responsibility distribution**. JavaScript frameworks typically assume a client-driven UI runtime with the server as an API provider. Wt assumes a server-driven application runtime where UI state and behavior live in a long-running C++ process, and the browser acts as a rendering and event endpoint.
This difference affects every major engineering dimension: state ownership, architecture, toolchains, scaling assumptions, security boundaries, performance tradeoffs, and maintainability. The purpose of this book is not to declare one model superior in all cases, but to teach Wt as a deliberate, production-grade alternative—one that aligns naturally with modern C++ engineering values when the problem domain justifies it.

1.3 When Wt is the right choice — and when it is not

1.3.1 The decision is architectural, not ideological

Choosing Wt is not primarily a question of whether C++ can be used for the web. It is a question of whether the *server-driven, stateful application model* provided by Wt matches the true engineering constraints of your system.
Many teams adopt technologies by habit: client-heavy frameworks for anything web-related, or C++ only for low-level components. Professional design requires the opposite: begin from constraints, then select an architecture that minimizes risk over the system's full lifecycle.
Wt is the right choice when its model reduces complexity and increases correctness. It is the wrong choice when it introduces avoidable operational cost, mismatches the UI requirements, or forces a stateful architecture where statelessness is a better fit.

1.3.2 A practical checklist for “right choice”

Wt is typically a strong choice when most of the following statements are true.

You need server-side authority over UI behavior

Wt excels when the UI is not merely a view over data but a controlled workflow:

- multi-step forms with strong validation and complex transitions,
- admin consoles where correctness and policy enforcement matter,
- internal tools where business rules must be enforced consistently,
- systems where UI state must remain consistent with domain state.

Because the authoritative UI state and event handling live on the server, the system naturally converges toward a single source of truth.

The application is inherently stateful

Wt is well suited for applications that are naturally modeled as stateful sessions:

- authenticated users with session context,
- dashboards that reflect live operational state,
- long-lived workflows and user-specific UI state,
- systems where “where the user is” is a meaningful part of correctness.

In such systems, forcing strict statelessness often moves complexity elsewhere (client caches, distributed session stores, duplicated workflow logic). Wt keeps state where it can be audited and controlled.

You need C++-level control, integration, or performance predictability

Wt becomes compelling when the web layer must integrate tightly with existing C++ infrastructure:

- reuse of mature C++ libraries and domain code,
- integration with embedded or industrial systems,
- heavy computation, simulation, or low-latency pipelines,
- environments where predictable resource usage matters.

The advantage here is not only speed, but also engineering leverage: strong typing, RAII, and shared code across components.

Long-term maintainability is a primary objective

Wt is often chosen in environments where the true cost is not initial development but years of maintenance:

- stable teams that value disciplined architecture,
- regulated environments and audit requirements,
- systems expected to run for many years with controlled evolution,
- projects that prioritize clarity, observability, and correctness.

A server-centric model can reduce cross-language duplication and drift, keeping business rules unified.

Your deployment model supports managed services

Wt applications are long-running processes. They fit naturally when your operations model already supports:

- reverse proxies (for example Nginx),
- service supervision (for example systemd),
- structured logging and monitoring,
- controlled rollout and restart strategies.

1.3.3 Typical domains where Wt is a strong fit

The following domains frequently align with Wt's strengths:

- **Internal enterprise applications:** administrative portals, management dashboards, workflow systems.
- **Operations and control systems:** monitoring, configuration, device management, industrial interfaces.
- **Government and institutional systems:** long-lifecycle software with strict policies and stable operational requirements.
- **Data-intensive back-office tools:** large tables, filtering, role-based access control, strong validation.
- **C++-centric organizations:** teams with strong C++ expertise and existing C++ code to leverage.

These systems often benefit from centralized correctness, explicit state management, and unified tooling.

1.3.4 When Wt is not the right choice

Wt is not a universal solution. It is typically a poor fit when one or more of the following conditions dominate.

You primarily need a content website or static marketing pages

If the system is mostly static content (landing pages, simple marketing sites, blogs, documentation sites), Wt is usually unnecessary. Static generation or conventional web stacks will be simpler, cheaper to host, easier to deploy, and easier for non-C++ contributors to maintain.

The UI demands a highly client-native experience

When the user experience is dominated by client-side requirements, such as:

- offline-first behavior and local-first storage,
- heavy client-side rendering and complex animations,
- rich media timelines, canvases, or large interactive front-end logic,
- highly dynamic SPA behavior that must remain responsive under high latency,

a dedicated client-heavy architecture may be more appropriate. Wt can support rich interactivity, but its server-driven nature makes it less ideal when the application model is meant to live in the browser.

You require extreme horizontal scaling with stateless service assumptions

If your architecture is built around strict statelessness and high horizontal scale where instances are freely interchangeable, a server-driven session model introduces constraints:

- you must manage session affinity or session replication,
- memory usage grows with active sessions,
- scaling strategies must account for stateful user contexts.

This is not a blocker in itself, but it changes the operational model. If your primary goal is maximal elasticity with minimal coordination, a stateless API plus client-managed UI may be a better fit.

Your team and organization are not prepared for C++ web operations

Even when Wt is technically suitable, it becomes a poor choice if the organization cannot support its operational needs:

- lack of C++ build expertise (toolchains, ABI considerations, dependency management),
- weak production discipline (monitoring, logging, service management),
- no appetite for compiled-service deployment pipelines,
- a staffing model centered on frontend developers rather than C++ engineers.

In such environments, the risk is not that Wt is flawed; the risk is that the organization will under-invest in the operational practices required to run a C++ web service responsibly.

Rapid UI iteration by non-C++ contributors is a primary requirement

If the product expects very fast UI iteration by designers and frontend developers who do not write C++, a JavaScript-centric approach will often be more efficient. Wt can be productive for C++ developers, but it does not provide the same ecosystem fit for teams built around client-side workflows and toolchains.

1.3.5 Decision framework: choose the model that reduces total system risk

A professional decision can be made by evaluating total system risk across five dimensions:

1. **Correctness and invariants:** where can the system most reliably enforce rules?
2. **Complexity distribution:** will complexity be centralized or duplicated across runtimes?
3. **Operational model:** can the organization run and monitor a long-lived service correctly?
4. **Performance and predictability:** are latency, throughput, and resource limits critical?
5. **Team capability and lifecycle:** will the system be maintained by C++ engineers over time?

Wt is typically justified when centralized correctness, maintainability, and C++ integration reduce risk more than the operational cost of a managed server process increases it.

1.3.6 A concise rule of thumb

> Use Wt when your web application is best modeled as a long-running, stateful C++ program with strict correctness and operational requirements. Prefer client-heavy JavaScript stacks when the browser is the natural home of the application model, rapid UI iteration dominates, or stateless horizontal scaling is the primary objective.

1.3.7 Summary

Wt is the right choice when it aligns with your system's true constraints: stateful workflows, server-side authority, C++ integration, long-term maintainability, and a deployment model that supports managed services. It is not the right choice for static content sites, UI experiences

dominated by client-side logic, organizations unprepared for compiled-service operations, or architectures that require strict stateless scaling by default. The engineering task is not to defend a technology; it is to select the model that minimizes total complexity and maximizes correctness for the real production environment.

1.4 Wt as a server-side web platform

1.4.1 Platform meaning: more than a UI library

Calling Wt a *server-side web platform* is not a marketing phrase. It describes a concrete engineering reality: Wt provides an integrated runtime model for building interactive web applications as long-running C++ server processes, including a structured UI composition model, an event-driven interaction mechanism, session-aware state management, and the infrastructure needed to expose the application over HTTP and WebSockets.
In this sense, Wt is not simply a widget collection. A widget collection would be an API surface for rendering UI. Wt goes further by defining:

- how an application is instantiated and hosted,
- how sessions are created and maintained,
- how browser events are transported to server handlers,
- how server-side state changes are synchronized back to the browser,
- how resources (CSS, images) are served,
- how deployment can be performed as a supervised service.

A “platform” is the combination of an application model and the operational runtime that supports it. Wt provides both.

1.4.2 The hosting model: a long-running C++ service

The primary deployment unit of a Wt application is a compiled executable that runs as a server process. This process listens for incoming HTTP connections and creates application sessions as users connect.

A minimal hosting skeleton illustrates the platform nature: you do not write handlers for individual endpoints as the primary abstraction; instead, you register an application entry point and let the framework host it.

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WServer.h>
#include <memory>

class App final : public Wt::WApplication {
public:
  explicit App(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Wt Platform Skeleton");
  }
};

int main(int argc, char** argv)
{
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);

  server.addEntryPoint(Wt::EntryPointType::Application,
    [](const Wt::WEnvironment& env) {
      return std::make_unique<App>(env);
```

```
    });

  return server.run();
}
```

This is closer to hosting a GUI program than to deploying a template-based website. The application is a program with life, state, and behavior managed over time.

1.4.3 Sessions as first-class runtime entities

In many web stacks, sessions are an add-on: a cookie plus server storage or a token plus database. In Wt, sessions are native to the platform model. The system is designed around the idea that each connected user interacts with an application instance that can hold state.
From an engineering perspective, sessions in Wt provide:

- a natural place for user context (authentication state, roles, preferences),
- a structured lifetime for UI state (current page, active widgets, forms),
- a predictable boundary for per-user caches and temporary objects,
- an explicit scope for concurrency decisions and resource ownership.

Because sessions are central, the developer must treat them responsibly: memory usage scales with active sessions, and server-side state must be designed to remain bounded and correct. This is precisely why Wt should be treated as a platform: it provides the runtime entity, but professional engineering must define how to use it safely.

1.4.4 The widget tree as an application model

Wt exposes a widget-based UI model in C++. A widget tree is not only a rendering tool; it becomes part of the application state. Widgets may hold user input values, validation results, visibility state, and domain-specific view state.

The platform value here is that UI state becomes:

- typed and structured (C++ objects rather than dynamic structures),
- explicit in ownership and lifetime (RAII and deterministic cleanup),
- naturally integrated with server-side logic (services, controllers).

This provides a unified system design: UI and domain logic can be kept consistent without duplicating behavior across multiple runtimes.

1.4.5 Event-driven interaction: browser events to C++ handlers

Wt applications are interactive because the framework transports browser events to server-side C++ event handlers and then synchronizes the resulting UI changes back to the browser.
The platform supplies:

- an event abstraction (signals) for user interactions,
- a mechanism to queue, process, and serialize updates,
- a mapping from server-side widget changes to client-side DOM updates,
- support for real-time interaction channels when appropriate.

The developer writes application logic as event handlers, typically within the UI layer (for presentation logic) and delegated to controllers and services for domain logic. This encourages an architecture that resembles a structured desktop application while still delivering through the web.

1.4.6 Transport and synchronization: incremental updates

A key characteristic of Wt as a server-side platform is its ability to send incremental updates to the client rather than re-render full pages. The user interacts, the server updates widget state, and only the required changes are transmitted to reflect the new UI state.
This model has several practical implications:

- complex multi-step workflows can remain consistent and controlled server-side,
- validation and UI feedback can be centralized without duplicating rules,
- the browser receives just enough information to update the interface,
- the system naturally supports a stateful interaction style.

In professional terms, incremental synchronization is an engineering mechanism that enables interactive experiences while keeping authority and logic on the server.

1.4.7 Resources and theming: integrated web concerns

Wt applications usually serve more than dynamic UI changes. They also include static resources: CSS files, images, icons, and sometimes downloadable reports. As a platform, Wt integrates these concerns within the application environment and deployment model.
From a system design perspective, this means:

- resources can be packaged and deployed alongside the application,
- URL routing and resource access are managed consistently,
- theming can be applied centrally and reproducibly.

A production-grade system should treat static resources as part of the deployable artifact, not as an informal collection of files. Wt encourages this integration by design.

1.4.8 Integration with the C++ ecosystem

A server-side platform must support the reality of production software: existing libraries, domain code, and infrastructure integration. Wt's value in C++ organizations is that it can directly integrate with:

- domain libraries and algorithms already written in C++,
- database layers and serialization systems designed for C++,
- performance-critical components with predictable behavior,
- mature C++ testing, profiling, and diagnostics workflows.

This reduces the friction of introducing the web layer into a C++ system: instead of translating domain logic into another language for UI behavior, Wt allows a unified codebase and a unified tooling culture.

1.4.9 Operational deployment: reverse proxy and service management

Treating Wt as a platform implies treating deployment as part of the engineering system. A typical production approach is to run the Wt executable as a service behind a reverse proxy (commonly Nginx). The reverse proxy handles:

- TLS termination (HTTPS),
- static caching policies where appropriate,
- request routing and virtual hosts,
- connection management and WebSocket forwarding.

The Wt process is then supervised by a service manager (commonly `systemd`), enabling:

- automatic restarts on failure,
- controlled startup order and resource limits,
- centralized logging and operational diagnostics.

This operational pattern is not optional decoration; it is part of the platform story. The application is a managed service, and the platform approach requires production discipline.

1.4.10 A platform-oriented architecture: layered responsibility

To use Wt as a platform professionally, the application must be architected as a system, not as a single file of widgets. A practical layered design often looks like:

- **UI layer:** widgets, layouts, user interaction wiring.
- **Controller layer:** orchestration between UI and services.
- **Service layer:** domain rules and business logic.
- **Data layer:** database access, transactions, persistence boundaries.

Wt makes this separation achievable because it centralizes the application runtime; the developer can enforce architectural boundaries within one language, one build system, and one deployment artifact.

1.4.11 Summary

Wt is a server-side web platform because it provides a complete runtime and application model for building interactive web systems in C++. It hosts applications as long-running processes, treats sessions and state as first-class concepts, expresses UI through a typed widget tree, and synchronizes browser interaction through an event-driven server model. In production, it

integrates naturally with reverse proxies and service supervision, and it supports disciplined architecture and operational clarity.
The value of Wt is not merely that it can generate web pages. Its value is that it enables a class of web applications to be engineered as structured, maintainable, and deployable C++ systems, with server-side authority and explicit control over state, lifetimes, and behavior.

1.5 Real-world use cases (Enterprise, Government, Internal Systems)

1.5.1 Why these domains align with Wt

Wt is most compelling in domains where a web application must be engineered as a long-lived system rather than treated as a short-term product experiment. Enterprise, government, and internal systems tend to share stable constraints:

- long operational lifecycles measured in years,
- strict control over authentication, authorization, and auditing,
- complex workflows with non-trivial validation and policy rules,
- integration with existing infrastructure (databases, directories, legacy systems),
- operational requirements such as predictable behavior, controlled deployment, and traceability.

These constraints align with Wt's server-driven model because the application can remain a single, strongly typed, auditable C++ codebase deployed as a managed service. The platform enables centralized enforcement of invariants and policies, and it naturally supports structured UI workflows that remain consistent under change.

1.5.2 Enterprise applications

Enterprise software is defined less by technology choice and more by operational reality: thousands of users, role-based access control, policy enforcement, and continuous evolution under note-worthy risk. Wt fits particularly well in back-office systems and administrative applications where correctness, control, and maintainability are primary goals.

Administrative consoles and operational dashboards

A common enterprise need is a secure administrative portal used by staff rather than the public. Such systems typically require:

- authenticated access with roles and fine-grained permissions,
- controlled workflow transitions (approval, escalation, review),
- large data tables with filtering, paging, and export,
- audit trails and operational logging,
- predictable performance under concurrent usage.

Wt supports these naturally because UI state and domain rules remain on the server. Validation rules can be enforced once, and the UI can reflect policy decisions immediately without duplicating logic in the browser.

Workflow-driven internal tools

Enterprise systems often encode workflows: sequences of steps with constraints, branching logic, and requirements for supervisor approval. Examples include procurement flows, inventory adjustments, compliance reviews, and staff onboarding. These systems benefit from:

- a stateful interaction model (the user's current step and context matter),

- server-side enforcement (preventing bypass of rules),
- explicit lifecycle control (transactions, rollback, and clear failure modes).

Wt's application-per-session model supports this well. The system can hold per-user workflow context in memory when appropriate, commit state to the database with transactional discipline, and maintain a coherent UI that mirrors business state.

Systems with strong integration requirements

Large organizations rarely start from zero. They integrate with:

- existing databases and schema constraints,
- directory services and identity providers,
- internal services and message buses,
- legacy C++ libraries and domain code.

Wt is advantageous when C++ is already present in the organization. The web layer can reuse domain libraries directly, reducing translation layers and decreasing the risk of semantic drift between "frontend" rules and "backend" rules.

1.5.3 Government and institutional systems

Government and institutional systems typically differ from consumer web products in three primary ways:

- the systems must remain stable and maintainable over long periods,
- the systems must support auditing, accountability, and documented processes,

- the systems often operate under strict security constraints and controlled environments.

For such systems, the choice of architecture often matters more than the choice of language. Wt can be a strong fit because it supports a server-centric model where policy enforcement and workflow control are centralized, and the operational deployment can be treated as a supervised service with predictable behavior.

Case management and process tracking

Institutional systems often manage cases: requests, incidents, permits, disputes, or investigations. These systems require:

- role separation (citizen, clerk, supervisor, auditor),
- controlled transitions (submitted, reviewed, approved, rejected),
- traceability of actions and data changes,
- strong validation aligned with regulation and policy.

Wt's architecture encourages such controlled transitions because event handling is server-side, and the UI workflow can be built as a structured program rather than scattered across distributed client logic.

Regulated data and controlled environments

Many government deployments occur in controlled networks with strict operational rules:

- locked-down client environments,
- restricted dependencies and whitelisting,
- long update cycles,

- strict compliance requirements for data handling.

In such environments, reducing moving parts is an advantage. Wt can reduce dependency fragmentation by keeping application behavior centralized in a compiled service. Operational updates become controlled deployments of a server artifact rather than frequent and complex multi-toolchain client releases.

Long-term maintainability as a primary requirement

Institutional software often outlives teams. A server-driven, strongly typed, single-language system can be easier to maintain when requirements evolve slowly but persistently. With disciplined layering, Wt applications can remain stable under growth: new forms, new workflows, new roles, and new reporting requirements.

1.5.4 Internal systems and private platforms

Internal systems are often where Wt provides its strongest practical value. These systems are not judged by public web trends; they are judged by operational reliability, integration, and the ability to serve employees efficiently.

Monitoring and control panels for infrastructure

Organizations operate fleets of systems: devices, services, data pipelines, and industrial components. They need control panels that provide:

- live status views and operational metrics,
- controlled actions with authorization,
- safe parameter changes and configuration workflows,
- logging of operational actions.

Such tools are frequently stateful and policy-driven. Wt can model these interactions cleanly as event-driven UI flows backed by server-side domain logic.

Back-office reporting and data exploration

Internal reporting systems often involve complex queries and large datasets. Typical features include:

- multi-criteria filtering,
- paging and sorting,
- exporting to files,
- user-specific saved views,
- controlled access to sensitive fields.

Wt supports this pattern well when combined with a disciplined data layer, because the server controls query execution and can enforce security constraints at the point of data access.

Enterprise “glue” systems

Many internal systems exist to connect other systems:

- data synchronization tools,
- account provisioning consoles,
- configuration and policy distribution tools,
- migration dashboards.

These tools are often small in scope but high in consequence. The benefits of strong typing, explicit resource control, and centralized logic are amplified because mistakes have operational impact.

1.5.5 Characteristics that repeatedly appear in successful Wt deployments

Across enterprise, government, and internal domains, the successful use cases tend to share these traits:

1. **Workflow significance:** the user journey is structured, and server-side enforcement matters.
2. **Policy and roles:** authorization is fine-grained and must remain consistent.
3. **Operational consequences:** actions have real impact and must be traceable.
4. **Integration depth:** the application integrates with existing systems and libraries.
5. **Lifecycle length:** maintainability over years is more important than short-term trend alignment.

These are precisely the conditions under which a server-driven C++ platform can reduce total complexity and increase confidence in correctness.

1.5.6 Where these use cases fail if the architecture is wrong

It is important to recognize that many internal systems fail not because the chosen language is wrong, but because responsibilities are mixed. In the context of Wt, the most damaging failure patterns include:

- placing database operations directly inside UI event handlers,
- mixing business rules with widget wiring,
- allowing session state to grow without bounds,
- deploying without supervision, logging discipline, or controlled configuration.

The rest of this book treats these as design constraints, and it demonstrates a layered architecture intended to prevent them.

1.5.7 A small concrete scenario: role-based administrative workflow

The following snippet illustrates the type of server-side policy enforcement that appears frequently in real internal systems: a role-gated action whose availability and execution are controlled on the server.

```
enum class Role { Viewer, Operator, Admin };

struct UserContext {
  Role role{Role::Viewer};
};

static bool canApprove(const UserContext& u) {
  return u.role == Role::Admin;
}

// In a controller/service boundary, not inside the UI layer:
static void approveChange(const UserContext& u) {
  if (!canApprove(u)) {
    throw std::runtime_error("Forbidden: approval requires Admin role.");
  }
  // Perform transactional state change here.
}
```

This is representative of why these domains favor server-driven architectures: enforcement is centralized, auditable, and independent of client-side behavior.

1.5.8 Summary

Enterprise, government, and internal systems often require a stable, auditable, policy-driven application model with strong integration and long-term maintainability. Wt aligns with these realities by enabling web applications to be engineered as long-running C++ services with server-side authority over UI state, workflows, and enforcement. In practice, Wt is most successful where the web UI is a controlled interface to consequential operations, and where disciplined architecture and operational clarity are treated as first-class engineering requirements.

How Wt Works Internally

2.1 Application lifecycle

2.1.1 Overview

Understanding the application lifecycle in Wt is essential for building robust server-side web applications. Unlike traditional stateless web frameworks where a new process or thread handles each request independently, Wt applications are long-lived server processes that manage user sessions and interaction state explicitly. The application lifecycle defines how the server starts, how user sessions are created and destroyed, how events are processed, and how resources are cleaned up at the end of a session or the entire application lifetime.
This section provides a systematic, detailed exposition of the Wt application lifecycle, including initialization, session creation, event handling, and termination.

2.1.2 Process startup and configuration

The first stage of the lifecycle is process startup. A Wt application is compiled as an executable that is intended to run as a server. It is not a passive library loaded per-request, but an active process that listens for incoming connections. The typical entry point for Wt applications is a `main()` function that creates and configures a `Wt::WServer` instance.
During startup, the following tasks occur:

- The server instance is configured with command-line arguments or configuration files.
- The HTTP listener and optional WebSocket support are initialized.
- Application entry points are registered through callbacks.
- Static resources such as CSS and JavaScript are prepared to be served.
- Logging and diagnostic subsystems are initialized.

At this stage, no user-specific state exists. The server is ready to accept connections and instantiate application sessions as users arrive.

```
int main(int argc, char** argv)
{
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);

  server.addEntryPoint(Wt::EntryPointType::Application,
    [](const Wt::WEnvironment& env) {
      return std::make_unique<MyApplication>(env);
    });

  return server.run();
}
```

In this example, the server is configured once at startup, and an entry point is registered. The callback constructs instances of a custom `MyApplication` class for each user session.

2.1.3 Session creation and request processing

Once the server is running and listening on a network port, client connections are accepted. For each new browser session, Wt creates a new instance of `WApplication`. This instance acts as the root of the widget tree and encapsulates the state of a single user interaction context.

The creation of a session involves:

- Parsing the incoming HTTP request and environment values.
- Allocating memory for the new session object.
- Initializing UI components as defined in the application constructor.
- Sending an initial response to the client with the rendered HTML and embedded state references.

At this stage, the session exists as a long-lived object with a distinct identity in the server process. It will persist across multiple HTTP requests or WebSocket messages until explicitly terminated or expired due to inactivity.

2.1.4 Widget tree initialization

Within each session, the `WApplication` instance constructs a widget tree. The widget tree represents the UI hierarchy, where each widget is a C++ object with clearly defined ownership and lifetime management.
Widgets may include:

- Containers to organize layout.
- Inputs for forms and user interaction.
- Data display elements such as tables and lists.
- Custom widgets defined by the application.

Widget initialization typically occurs in the application constructor or within factory methods invoked during session creation.

```
class MyApplication : public Wt::WApplication {
public:
  MyApplication(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Application Lifecycle Demo");
    root()->addWidget(std::make_unique<Wt::WText>("Welcome"));
  }
};
```

The widget tree becomes the authoritative representation of the UI state. This tree persists for the duration of the session, and all server-side event handlers operate on this structure.

2.1.5 Event handling and user interaction

User interaction in Wt is event-driven. When a user interacts with the browser (clicks a button, submits a form, changes a selection), events are forwarded to the server either via HTTP requests or WebSocket messages. The Wt platform handles the transport of these events to the appropriate widget event handlers.
The lifecycle of event handling includes:

- Client-side event serialization.
- Server-side event deserialization and dispatch.
- Execution of the corresponding C++ event handler in the context of the widget tree.
- Modification of the widget state and/or application state.
- Generation of a minimal set of changes that need to be sent back to the client.
- Transmission of these incremental updates to the browser.

This mechanism enables interactive applications that feel responsive without requiring full-page refreshes. The server retains control of the application logic, while the client reflects changes as needed.

2.1.6 Session persistence, timeout, and cleanup

Sessions in Wt can persist for long periods, particularly when the user is actively interacting with the application. However, production systems must not allow sessions to live indefinitely without bounds. Therefore, session timeout and cleanup are essential aspects of the lifecycle. A session may be terminated in several ways:

- Explicit user logout.
- Session inactivity timeout configured on the server.
- Server shutdown or restart.
- Explicit session destruction initiated within application logic.

When a session ends, the Wt runtime ensures that:

- All widgets in the session are destroyed in a deterministic order.
- Resources owned by the session (memory, handles, caches) are released.
- Any required teardown logic in destructors is executed.

This deterministic cleanup is critical for memory correctness and avoids resource leaks in long-running deployments.

2.1.7 Handling long-running processes and concurrency

Wt applications run as server processes that may handle multiple sessions concurrently. The concurrency model in Wt is cooperative: each sessions event handler runs to completion before another event is processed in the same thread. Multiple threads can be configured, each managing its own set of sessions and events.
Concurrency is important in the lifecycle because:

- Multiple users can interact with the application simultaneously.
- Sessions must be isolated from each other.
- Shared resources (database connections, caches) must be protected with appropriate synchronization.

The developer must design services and controllers to handle concurrent access correctly, using locks or other synchronization primitives as appropriate.

2.1.8 Application shutdown and cleanup

Eventually, the server process will be stopped for maintenance, upgrade, or redeployment. Wt provides mechanisms for graceful shutdown:

- No new sessions are accepted.
- Existing sessions are given an opportunity to complete in-flight operations.
- The widget trees and session contexts are safely torn down.
- System resources are released before the process exits.

A graceful shutdown is vital in production systems to avoid losing in-flight user activity and to ensure that external resources (such as database connections) remain consistent.

2.1.9 Lifecycle observability and diagnostics

In production environments, understanding the state of the application lifecycle is essential for operational health. Wt runtime can be instrumented to provide:

- logging of session creation and destruction,
- metrics on active sessions and request latency,
- error reporting for event handler exceptions,
- integration with external monitoring systems.

These observability features help in capacity planning and debugging.

2.1.10 Summary

The application lifecycle in Wt consists of:

1. **Server process startup and configuration,**
2. **User session creation and widget tree initialization,**
3. **Event-driven interaction and state updates,**
4. **Session timeout and cleanup,**
5. **Concurrent session management,**
6. **Graceful shutdown and resource release,**
7. **Operational observability of the application state.**

Understanding this lifecycle is crucial to building scalable, responsive, and maintainable Wt applications. It shapes how resources are allocated, how user interactions are managed, and how the server behaves under load and during maintenance operations.

2.2 One `WApplication` per session

2.2.1 The central runtime model

A defining internal property of Wt is that it creates one `Wt::WApplication` instance for each user session. This is not a minor implementation detail; it is the foundation that explains how Wt can behave like a server-side GUI framework while still delivering a web user interface.
In practical terms, the model can be summarized as:

- The server process is long-running and shared by all users.
- Each user session is represented by a distinct `WApplication` object.
- Each `WApplication` owns a widget tree that represents that user's UI state.
- Browser interactions are routed to the corresponding session, and therefore to the corresponding `WApplication` instance.

The result is a clean separation between users: your application code can store per-user UI state and per-user workflow context in ordinary C++ objects with ordinary C++ lifetimes.

2.2.2 What "session" means in Wt

In Wt, a session is the server-side interaction context associated with a browser visiting your application URL. Conceptually, it is the entity that allows Wt to maintain state across multiple requests and interactions.
A session typically includes:

- the `WApplication` instance and its widget tree,
- per-session environment information derived from `WEnvironment`,

- event queues and update synchronization state,
- server-side session variables and user context,
- optional application-level caches that are safe and bounded per user.

Sessions eventually end (for example when the user leaves, the session times out, `WApplication::quit()` is called, or the server shuts down). When a session ends, the corresponding `WApplication` instance and its widget tree are destroyed, and destructors run deterministically.

2.2.3 How the framework creates a `WApplication` instance

The creation point is explicit in Wt: you register an entry point with `WServer` (or use a helper such as `WRun()`), and provide a factory that constructs a `WApplication` for each session.

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WServer.h>
#include <memory>

class App final : public Wt::WApplication {
public:
  explicit App(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("One WApplication per session");
    // Build the widget tree for this session here.
  }
};
```

```
int main(int argc, char** argv)
{
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);

  server.addEntryPoint(Wt::EntryPointType::Application,
    [](const Wt::WEnvironment& env) {
      // Called once per user session:
      return std::make_unique<App>(env);
    });

  return server.run();
}
```

The critical property is that the factory lambda is invoked per session, meaning each user receives a separate App object and a separate widget tree.

2.2.4 Consequences: isolation, state, and correctness

Because each session has its own `WApplication` instance, the following design becomes natural and safe:

- per-user UI state (selected tab, form fields, view filters) lives in member variables,
- per-user workflow context (current step, validation state, pending operations) lives in session-owned objects,
- per-user authorization context (role, permissions, tenant) can be stored once and used consistently,
- per-user caching can be implemented with explicit bounds and lifetimes.

This is a major reason Wt is attractive for internal and workflow-heavy systems: server-side correctness is easier to enforce when the authoritative interaction state is server-owned and session-scoped.

2.2.5 Widget tree ownership and lifetime

Each `WApplication` instance owns a widget tree that represents the UI for that session. When you build UI in the constructor (or in functions called by the constructor), you are allocating objects that belong to that session.
In a disciplined architecture, it is useful to remember this rule:

> If a widget is reachable from `root()`, it is session-owned and will be destroyed when the session ends.

Therefore, session lifetime and widget lifetime are tightly coupled. This is a strength because it provides deterministic cleanup, but it also places responsibility on the developer to avoid holding unsafe references to widgets outside the session lifetime.

2.2.6 Shared state vs session state: what must not be confused

The "one `WApplication` per session" model can mislead developers into one of two mistakes:

1. treating session state as if it were application-global state,
2. treating application-global resources as if they were session-local.

A professional Wt system normally has two categories of state:

Session-local state (belongs to one user)

This belongs inside `WApplication` or inside session-owned objects:

- logged-in user identity and roles,
- per-user UI state and preferences,
- per-user workflow state and temporary buffers.

Shared process-wide resources (used by many sessions)

These must be carefully designed for concurrency and lifetime:

- database connection pools,
- immutable configuration objects,
- shared caches with explicit synchronization and eviction,
- logging infrastructure.

A common professional pattern is to keep shared resources in a separate "application services" object that is created at process startup and then referenced by each session through safe, controlled interfaces.

2.2.7 Thread-safety implications

Wt isolates sessions in terms of object graphs, but it does not automatically make arbitrary code thread-safe. The server can handle multiple sessions concurrently, which means that any process-wide shared resources must be safe under concurrent access.
Within a single session, UI manipulation is normally performed within Wt's event handling context. When you introduce background work (threads, async tasks, timers), you must treat the session lifetime and UI thread-safety rules as hard constraints.

A safe mental model is:

- Session UI state is not a free-for-all shared object graph.
- Modifying widgets from outside the normal event context must be done using the framework's supported mechanisms for safe server-side updates, and only while the session is still alive.

In other words, session isolation is about ownership and lifetime, not about bypassing concurrency discipline.

2.2.8 Why this model changes how you design your architecture

Because each session is an application instance, the temptation is to put everything in `WApplication`. That approach works only for demos. For production systems, you must keep `WApplication` thin and delegate responsibilities.
A practical layered architecture compatible with this model is:

- **WApplication:** session setup, root layout, navigation wiring, session context.
- **Views (widgets):** UI composition, presentation logic, event wiring.
- **Controllers:** orchestration between UI events and domain operations.
- **Services:** business rules, validation policies, authorization checks.
- **Data layer:** transactions, persistence, repositories, database access.

This design keeps session-owned objects clear and bounded, while shared resources remain centralized and carefully controlled.

2.2.9 A small, concrete pattern: session context

A useful technique is to formalize session-owned identity and authorization context as a small object that lives inside `WApplication`:

```
struct SessionContext {
  bool authenticated{false};
  std::string userId;
  std::string displayName;
  // Add role/permission data as needed, kept per session.
};

class App final : public Wt::WApplication {
public:
  explicit App(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    // Initialize UI and session context for this user session.
  }

  SessionContext& session() noexcept { return session_; }
  const SessionContext& session() const noexcept { return session_; }

private:
  SessionContext session_;
};
```

This keeps session state explicit and avoids accidental mixing of session and shared state.

2.2.10 Memory and capacity planning

One `WApplication` per session implies that server memory usage grows with the number of active sessions and with how much state you retain per session. This has direct production consequences:

- per-session widget trees should be kept reasonable in size,
- per-session caches must be bounded,
- large datasets should not be stored in session memory unless required and controlled,
- long-lived background work must not retain session-owned pointers after session termination.

A professional Wt application treats session memory as a capacity-managed resource, just like file descriptors or database connections.

2.2.11 Termination and deterministic cleanup

When a session ends, Wt destroys the `WApplication` instance and its widget tree. This provides deterministic cleanup via C++ destructors. Proper production code uses destructors to release session-owned resources:

- cancel timers and background tasks tied to the session,
- release per-session caches and buffers,
- ensure no background thread can outlive the session while holding unsafe references.

The deterministic cleanup property is a major advantage of this model, but it only remains safe when the application avoids dangling references and properly coordinates background work.

2.2.12 Summary

The “one `WApplication` per session” model is the cornerstone of Wt’s internal design. It provides per-user isolation through separate application instances and separate widget trees, enabling a stateful, event-driven interaction style implemented in C++. It also imposes real production responsibilities: session memory must be bounded, shared resources must be thread-safe, and background work must respect session lifetime and UI safety rules. When these constraints are handled with discipline, the model yields highly maintainable server-side web systems with centralized correctness and clear operational behavior.

2.3 Event loop and request handling model

2.3.1 Introduction

An essential technical foundation of Wt is its event loop and request handling model. Unlike traditional stateless web frameworks that treat each HTTP request as an independent transaction, Wt applications are architected as long-lived server processes that manage continuous interaction with clients. Understanding Wt’s event loop and request handling model is critical for designing responsive, correct, and maintainable applications.
This section explores how Wt processes incoming requests and client events, how internal dispatching works, and how stateful interactions are orchestrated within the server runtime.

2.3.2 The server as an event-driven runtime

A Wt application runs as a server process that listens for network connections on one or more ports. Internally, the runtime is based on an event-driven model. This model efficiently manages:

- incoming HTTP requests,

- WebSocket messages for interactive sessions,
- timers and deferred operations,
- asynchronous I/O when used.

The core idea is that the application waits on network and event sources, dispatches events to handlers, updates internal state, and then returns minimal responses or incremental UI updates to connected clients.
The event loop enables Wt to multiplex many sessions and many event sources within one or more operating system threads.

2.3.3 Connection handling: HTTP and WebSockets

Wt supports both classic HTTP request/response and WebSocket-based interaction. The approach used for a given session depends on configuration and client capabilities.

HTTP request/response

In a basic setup, the client sends an HTTP request. The server accepts the connection, parses the request, identifies the target session based on a session identifier (often a cookie or URI parameter), and delivers the request to the application entry point or event handler.
The server then constructs an HTTP response containing:

- the rendered initial UI or updated UI fragments,
- any necessary headers,
- state markers necessary for subsequent requests.

The event loop continues to run after each request and accepts further requests.

WebSocket interaction

WebSocket integration enables more efficient interaction patterns. Instead of opening a new HTTP connection for every event, the client upgrades the HTTP connection to a WebSocket connection. This channel remains open and bi-directional, enabling:

- lower-latency event forwarding,
- reduced overhead compared to repeated HTTP,
- a natural stream for event messages and updates.

Under WebSocket mode, the event loop handles WebSocket messages as another event source. Events are deserialized, dispatched to the appropriate handlers, and then any required UI updates are sent back over the same channel.

2.3.4 The central event loop

At the core of Wt's internal architecture is a loop that waits for activity from registered event sources. These sources include:

- network sockets (HTTP and WebSocket),
- timers,
- asynchronous I/O callbacks,
- internal control signals (shutdown, session cleanup).

The loop has the following general responsibilities:

1. polling and waiting for any event source to become active,
2. dispatching the event to the appropriate subsystem (session, timer, or control),

3. running event handlers to completion,
4. scheduling outgoing responses or incremental UI updates,
5. returning to wait for the next event.

This pattern is similar to event-driven architectures found in GUI frameworks, message loop systems, and asynchronous servers. The advantage is efficient use of resources: the server does not block on a single connection or session, and can handle many sessions concurrently.

2.3.5 From request to session dispatch

When a network event arrives, the server runtime must determine which session it belongs to. This is usually based on a session identifier that is part of the URL or a session cookie managed by the framework.
Once the session is identified:

- the event is mapped to a session-specific context,
- event data (form submission, button click, etc.) is deserialized,
- the appropriate handler in the widget tree or application logic is invoked,
- the widget state may be updated based on the logic,
- the framework determines what parts of the UI have changed and need to be sent back.

This dispatch process ensures that the event is processed within the correct session context and widget tree.

2.3.6 Widget event handling

Within a session, the UI is composed as a tree of widgets. Each widget may register event handlers for specific events. For example, a button might register a handler for a click event, or a form might register a handler for form submission.
When an event arrives:

- the runtime identifies the target widget,
- the handler associated with the widget is invoked in the application code,
- application state may be modified,
- the widget tree is updated accordingly,
- the runtime computes a minimal set of changes that need to be reflected in the client UI.

The runtime is careful to manage exceptions and errors within handlers so that a single session error does not crash the entire server process.

2.3.7 Incremental UI updates

A major efficiency characteristic of Wt is its ability to send only incremental UI updates to the client rather than full-page reloads. After event handling:

- the runtime compares the new widget state with the previous state,
- it determines a minimal set of DOM changes,
- it packages these changes as a concise update message,
- the update message is sent to the client.

This behavior conserves bandwidth and improves responsiveness, especially in complex interactive applications.

2.3.8 Timers and internal events

Wt supports timers and scheduled events on the server side. These internal event sources are registered with the runtime and dispatched by the central event loop just like network events. Common uses for timers include:

- polling external systems,
- updating live dashboards,
- session timeouts and periodic housekeeping.

The event loop ensures that timer events are integrated seamlessly with user interaction events.

2.3.9 Asynchronous operations

While Wt does not enforce a specific asynchronous programming model, it supports integration with asynchronous tasks. For example, a database query or file I/O can be executed in a separate thread or asynchronous context.
The essential pattern is:

- schedule the asynchronous task,
- return control to the event loop,
- when the task completes, signal back to the event loop,
- handle results within the session context,
- apply any necessary UI state updates.

This keeps the main event loop responsive and prevents blocking on long-running operations.

2.3.10 Concurrency and thread management

Wt applications may be configured to handle multiple threads, each with its own set of sessions and events. When multi-threading is enabled:

- the server runtime manages thread pools,
- each thread runs an event loop that waits for activity on its assigned sessions,
- shared resources must be protected with synchronization primitives.

The developer must ensure that shared state (for example caches and database connection pools) is thread-safe. Per-session state, by contrast, is naturally isolated in separate `WApplication` instances.

2.3.11 Graceful shutdown and event loop termination

A professional deployment must support controlled shutdowns. When the server receives a shutdown signal:

- the event loop stops accepting new connections,
- existing sessions are given a chance to complete in-flight events,
- the event loop runs until all necessary cleanup actions are performed,
- the server stops listening on sockets and exits.

Graceful shutdown ensures that resource cleanup, logging, and persistent state updates occur consistently.

2.3.12 Summary

Wt's event loop and request handling model are central to its identity as a server-side web platform. The model integrates:

- efficient multiplexing of network and timer events,
- session-based dispatch of user interactions,
- incremental UI updates,
- support for asynchronous tasks and timers,
- thread-aware concurrency,
- deterministic cleanup and graceful shutdown.

Understanding this model is essential for building responsive, scalable, and maintainable Wt applications. It reveals how Wt unifies interactive web programming with server-driven, event-oriented architecture, and it provides the basis for higher-level design decisions explored in subsequent chapters.

2.4 Stateful web applications

2.4.1 What "stateful" means in the context of Wt

A web application is *stateful* when the server retains an explicit interaction context for each user across multiple requests, and when the correctness of the application depends on that context. In such a system, the server does not merely respond to isolated requests; it maintains a coherent model of:

- the user's current UI and workflow position,

- the user's identity, authorization, and preferences,
- temporary data collected across multiple steps,
- pending operations and their validation state,
- UI component state that would otherwise need to be rebuilt or re-derived repeatedly.

Wt is designed around stateful interaction as a primary model. The framework assumes that maintaining server-side state is not an accident but a deliberate engineering choice that simplifies certain classes of applications and enables a server-driven UI runtime.

2.4.2 Why state matters in real systems

Many real-world applications are not naturally expressed as independent request/response transactions. Instead, they contain workflows that require continuity:

- multi-step forms where earlier inputs constrain later options,
- approval processes where transitions depend on role and history,
- dashboards that preserve filtering and navigation context,
- configuration tools where changes are staged and reviewed,
- interactive data exploration where the UI state must remain consistent with permissions and domain invariants.

In stateless architectures, continuity must still exist, but it is typically reconstructed on each request by combining client-provided parameters, session tokens, cached data, and database state. That reconstruction can be correct, but it often spreads complexity across multiple layers. Wt takes a different approach: it maintains the interaction model server-side, and therefore can treat UI continuity as a first-class property.

2.4.3 The Wt mechanism: per-session application instances

The practical foundation of Wt's statefulness is the "one `WApplication` per session" model. Each user session is represented by an application instance with its own widget tree and its own C++ state. The server therefore has a natural place to store session-specific state without inventing a secondary state system.
This model implies:

- session state can live as ordinary member variables,
- state lifetimes are deterministic (RAII),
- per-user invariants can be enforced centrally,
- UI state and domain rules can remain synchronized in one runtime.

2.4.4 Kinds of state in a stateful Wt application

Professional design requires distinguishing types of state, because not all state should be treated equally.

Interaction state (UI and workflow state)

This state describes what the user is doing:

- current view, tab, or route,
- partially completed form values,
- validation errors currently displayed,
- wizard step and branching conditions,
- selected filters and pagination state.

This state is often transient and should remain bounded.

Identity and authorization state

This state describes who the user is and what they are allowed to do:

- authentication status,
- user identity and tenant context (if applicable),
- roles and permissions,
- security-relevant flags (for example “must re-authenticate”).

This state must be treated as authoritative on the server and must not be derived solely from client inputs.

Domain state

This is the persistent system state stored in a database or durable storage:

- orders, requests, and case records,
- configuration objects,
- audit trails,
- reports and business entities.

This state should not be duplicated blindly in session memory; it should be accessed through a disciplined data layer with transactions.

Caches and performance state

Caches exist to reduce repeated work:

- small per-session caches of derived values,
- shared caches with strict synchronization and eviction,
- precomputed summaries.

Caches must be bounded and designed to remain correct under concurrency.

2.4.5 The benefits of statefulness in Wt

When used with discipline, statefulness yields concrete engineering benefits.

Centralized correctness and single-source-of-truth behavior

Because UI logic and workflow control remain server-side, there is less pressure to replicate rules across client and server. Validation and policy checks can be executed consistently, and the UI can be updated directly based on authoritative decisions.

Natural modeling of complex workflows

A workflow can be represented as explicit states and transitions in C++ rather than as scattered client-side logic. This improves maintainability and makes it easier to test domain rules.

Deterministic lifetimes and resource safety

Session state is owned by C++ objects and cleaned up deterministically when the session ends. This allows the use of RAII for resources such as:

- temporary files,

- transaction scopes,
- locks and synchronization objects,
- session-scoped caches.

2.4.6 The costs and risks of statefulness

Statefulness is a design choice with operational and architectural consequences.

Memory usage scales with active sessions

Each active session retains some in-memory state. If session state grows unbounded, memory usage can become the limiting factor. Professional Wt applications therefore:

- keep widget trees and session caches bounded,
- avoid storing large datasets in session memory,
- release session resources promptly,
- use paging and on-demand loading for large tables.

Scaling requires deliberate strategy

Stateless services can be scaled by adding replicas without session affinity concerns. Stateful systems require additional planning:

- session affinity (routing a user to the same server instance),
- controlled session timeouts,
- careful design of shared resources,

- capacity planning based on concurrent active sessions.

This is not inherently difficult, but it is different from stateless assumptions and must be engineered.

Concurrency boundaries must be respected

Per-session state is isolated, but process-wide shared resources are not. Any shared caches, pools, or global registries must be thread-safe. In addition, background tasks must not access session-owned objects after a session ends.

2.4.7 A disciplined pattern: explicit session context

A professional stateful design makes session state explicit. The following pattern is widely useful: define a small `SessionContext` that captures identity and workflow essentials, and keep it inside `WApplication`.

```
#include <string>
#include <optional>

enum class Role { Viewer, Operator, Admin };

struct SessionContext {
  bool authenticated{false};
  std::string userId;
  std::string displayName;
  Role role{Role::Viewer};

  // Example of explicit workflow context:
  std::optional<std::string> activeCaseId;
};
```

This approach improves:

- readability (session state is not scattered),
- testability (services can receive a context object),
- security (authority is clearly server-owned),
- maintainability (adding new session properties is controlled).

2.4.8 A disciplined pattern: keep persistent state persistent

A common anti-pattern in stateful systems is to load large persistent domain objects into memory and keep them there for convenience. In production, this often causes memory growth, stale data, and concurrency problems.
A more robust approach is:

- store only identifiers and small derived values in session state,
- load domain objects when required,
- commit changes through explicit transactions,
- refresh views based on committed state, not cached guesses.

This maintains correctness and controls memory usage.

2.4.9 A small example: stateful multi-step workflow

The following example shows a minimal workflow state machine that could back a multi-step form. The point is not the completeness of the form, but the explicitness of workflow state and transitions.

```
enum class Step { Start, Details, Review, Done };

struct WizardState {
  Step step{Step::Start};
  std::string name;
  std::string email;

  void next() {
    switch (step) {
      case Step::Start:   step = Step::Details; break;
      case Step::Details: step = Step::Review;  break;
      case Step::Review:  step = Step::Done;    break;
      case Step::Done:    break;
    }
  }
};
```

In Wt, such state can be stored in the session and used to drive UI rendering and validation consistently on the server.

2.4.10 Operational implications: timeouts, cleanup, and bounded state

A stateful architecture must include lifecycle controls:

- session inactivity timeouts,
- deterministic cleanup on session end,
- bounded memory usage per session,
- logging of session creation and termination for operations.

These controls are not optional. They define the difference between a stateful system that remains stable and one that fails under load.

2.4.11 Summary

Wt is built around stateful web applications. It treats the user interaction context as a first-class server-side entity, represented by a per-session `WApplication` instance and an owned widget tree. This model enables centralized correctness, natural workflow modeling, and deterministic lifetimes through C++ semantics. However, it also imposes real engineering responsibilities: session state must remain bounded, scaling must be designed with statefulness in mind, and concurrency boundaries must be respected. When these constraints are handled professionally, statefulness becomes an advantage rather than a liability, especially in enterprise, government, and internal systems where correctness, policy enforcement, and long-term maintainability dominate.

2.5 HTTP vs WebSockets in Wt

2.5.1 Introduction

Wt supports two primary mechanisms for browserserver interaction: traditional HTTP request/response and persistent WebSocket connections. Both mechanisms are first-class citizens in Wt, and the framework uses them to deliver interactive web applications with efficient state synchronization, minimal overhead, and predictable behavior. Understanding the difference between these transport models, how Wt leverages them, and when to prefer one over the other is vital for building responsive, scalable, and maintainable applications.
This section provides a detailed, current, and academically grounded explanation of HTTP and WebSocket usage within Wt, how they are implemented, and how they influence architecture and performance.

2.5.2 The HTTP request/response model

The HTTP protocol is the foundation of the web. In the simplest Wt configuration, the client (web browser) initiates an HTTP request, the server processes it, and then the server sends a response. This model has several characteristics:

- **Stateless by design:** each request is independent at the protocol level.
- **Synchronous:** the client waits for the server to return a complete response.
- **Simple transport:** no persistent connection is required beyond the request.

Wt implements HTTP-based interaction in a stateful context. Even though HTTP itself is stateless, Wt sessions and application instances provide continuity across requests. When a page is first requested, the server responds with HTML, embedded state tokens, and references. Subsequent interactions can also be handled via HTTP requests that carry necessary session markers.

2.5.3 HTTP message lifecycle in Wt

When an HTTP request arrives:

1. The server accepts the connection and parses the HTTP headers.
2. Wt extracts session identifiers from cookies or URL parameters.
3. The appropriate `WApplication` instance is located based on the session.
4. The request is deserialized into an event or action.
5. Event handlers are invoked on the server.
6. The server computes UI updates or a complete response.

7. The response is sent back to the client.

This approach retains the classic request/response semantics while encapsulating continuity in the session context.

2.5.4 Limitations and overhead of HTTP-only interaction

While HTTP is reliable and compatible with existing infrastructure, it has certain limitations:

- **Latency:** each interaction may involve a full request and response cycle.
- **Resource overhead:** repeated requests require new HTTP headers, increasing network usage.
- **Concurrency:** rapid event sequences may incur noticeable network round-trips.

These limitations are not unique to Wt; they are inherent in HTTP. Wt mitigates them with incremental updates, but the overhead remains visible when rapid or frequent interactions occur.

2.5.5 WebSocket protocol: persistent and bi-directional

WebSocket is a protocol that upgrades an HTTP connection to a persistent, bi-directional communication channel. In Wt, WebSocket support allows the browser and server to exchange messages without the overhead of repeated HTTP requests.
The key characteristics of WebSockets are:

- **Single persistent connection:** the socket remains open for the duration of the session.
- **Bi-directional messages:** both client and server can send messages independently.
- **Low overhead:** after the initial handshake, messages omit HTTP headers.

- **Efficient for interactive applications:** rapid updates and event notifications are more efficient.

WebSockets are especially advantageous for high-frequency interaction, near-real-time updates, or applications that require minimal latency between client and server events.

2.5.6 WebSocket handshake and integration in Wt

The transition from HTTP to WebSocket in Wt follows a defined handshake:

1. The client requests an upgrade by sending a specific WebSocket handshake request to the server.
2. The server acknowledges and upgrades the connection.
3. A persistent channel is established.
4. Both sides use the channel to exchange event messages and UI updates.

Wt abstracts these details from the developer. Once the WebSocket is established, the framework treats it as an event transport layer for the same stateful session.

2.5.7 How Wt chooses the transport

Wt can automatically upgrade client connections to WebSocket when supported by:

- the browser,
- the server configuration,
- network proxies that allow WebSocket.

If WebSocket is not available, Wt gracefully falls back to conventional HTTP interaction while still maintaining session state and incremental UI updates.

2.5.8 Event transport and synchronization

Whether using HTTP or WebSocket, the core interaction pattern remains:

- client event (click, input change, form submit) is serialized,
- server event handler is invoked on the correct `WApplication` instance,
- UI state is updated,
- incremental changes are sent back to the client.

Under HTTP, this cycle involves separate request/response exchanges. Under WebSocket, the messages are exchanged on the same persistent channel, reducing overhead.

2.5.9 Incremental UI updates over WebSocket

One of the advantages of WebSockets in the context of Wt is that incremental UI updates are sent without the need for repeated HTTP response framing. After a WebSocket handshake:

- UI update messages are encoded concisely.
- The server pushes updates immediately after state changes.
- The client applies updates to the local DOM representation.

This model feels closer to a desktop GUI event loop where state changes are reflected immediately, and network framing overhead is minimal.

2.5.10 Production considerations: fallback, proxies, and compatibility

In real-world deployments, the network path between client and server may include reverse proxies, load balancers, and corporate firewalls. Some of these intermediaries may not support WebSocket upgrades. Wt's dual support ensures maximum compatibility:

- begin with HTTP connection setup,
- attempt WebSocket upgrade,
- if the upgrade fails, continue with HTTP interaction.

This fallback is critical for robustness in environments where WebSocket support is inconsistent.

2.5.11 Performance implications

The choice between HTTP and WebSocket affects performance in measurable ways:

- **HTTP-based interaction:** overhead per interaction is higher, useful for low-frequency events or simple applications.
- **WebSocket-based interaction:** overhead per interaction is lower, useful for high-frequency or real-time UI updates.

Latency, throughput, and resource usage vary accordingly, and choosing the right transport depends on interaction patterns.

2.5.12 Security considerations

Both HTTP and WebSocket transport can be protected with HTTPS/TLS. In production environments:

- TLS termination is commonly handled by a reverse proxy.
- Encrypted transport protects session identifiers and user data.
- Authentication tokens and cookies must be protected.

WebSocket messages under TLS maintain equivalent security properties to HTTPS. From an application perspective, the underlying transport does not change the security model, but network configuration must account for secure WebSocket (wss) support.

2.5.13 API implications for application developers

For most application code, Wt abstracts the difference between HTTP and WebSocket. Event handlers and widget logic remain the same. The choice of transport affects:

- perceived responsiveness,
- resource usage,
- network characteristics.

The developer configures the server and environment, and Wt handles message framing and session routing.

2.5.14 Monitoring and diagnostics

In production systems, visibility into transport usage can be important:

- server logs may indicate whether WebSocket handshake succeeded,
- middleware can expose metrics on message rates,
- connection counts and duration can be monitored.

This information can guide capacity planning and debugging.

2.5.15 Fallback and progressive enhancement

Wt's transport model can be seen as a form of progressive enhancement: provide full support for WebSockets when available, and gracefully degrade to HTTP-based interaction without loss of correctness or continuity.
This ensures that:

- older browsers remain functional,
- restrictive networks do not break core workflows,
- session state remains consistent regardless of transport.

2.5.16 Summary

In Wt, HTTP and WebSocket are complementary transport mechanisms. HTTP provides universal compatibility and stateless framing, while WebSocket offers persistent, low-overhead, bi-directional communication. Wt uses sessions and an event loop to interpret both transports in a stateful server context. Understanding how HTTP and WebSocket differ, how they are integrated, and how Wt abstracts transport details allows developers to optimize interaction patterns for responsiveness, performance, and operational robustness in production-grade applications.

Part II

Environment Setup and First Application

Development Environment Setup

3.1 Requirements on Linux and Windows

3.1.1 What you must have before you start

Building and developing with Wt is fundamentally a **native C++ build workflow**. Your environment must provide three core capabilities:

1. **A modern C++ compiler toolchain**
2. **CMake-based build configuration**
3. **Boost as the primary external dependency**

In addition, production-grade usage typically requires cryptography/TLS support, database client libraries (when using `Wt::Dbo`), and optionally graphics/text-rendering libraries if you enable related Wt features.

3.1.2 Minimum versions (portable baseline)

The following baseline is widely used and stable across platforms for building Wt from source:

- **CMake:** 3.13 or newer

- **Boost:** 1.66 or newer
- **Linux compilers:** GCC 8+ or Clang 7+
- **Windows compiler:** MSVC toolset 14.1+ (Visual Studio 2017 and newer toolsets)

This baseline targets a modern C++ implementation level suitable for production builds, reproducible CI, and clean dependency management.

3.1.3 Linux requirements

Core build tools

On Linux you need the standard compilation toolchain and build utilities:

- a C++ compiler (`g++` or `clang++`)
- `cmake`
- a build backend such as `ninja` or `make`
- `git` (recommended for source-based workflows)

Core Wt dependency

Wt depends on Boost, so you need the Boost development packages installed.

Common optional but highly recommended dependencies

Even when not strictly required to compile a minimal Wt build, production setups commonly enable these:

- **OpenSSL:** for HTTPS/TLS support in the built-in HTTP(S) server and secure deployments

- **zlib:** for compression-related features and common build configurations
- **Image/graphics and text stack (feature-dependent):**
  - GraphicsMagick (image processing and related functionality)
  - Pango + HarfBuzz (advanced text shaping/rendering features used by some deployments)
  - OpenGL libraries (if using Wt functionality that requires GL support)
  - libharu (PDF generation support, when enabled)
- **Database client libraries (when using `Wt::Dbo` backends):**
  - SQLite (`sqlite3`)
  - PostgreSQL client libraries
  - MySQL/MariaDB client libraries
  - Other enterprise DB backends are possible depending on your environment and configuration
- **FastCGI library (optional):** if you intend to deploy Wt behind a web server using FastCGI

Example installation sets (common distributions)

The exact package names differ by distribution. The following commands are **reference setups** that cover typical development needs.

Ubuntu/Debian (baseline + TLS + SQLite)

```
sudo apt update
sudo apt install -y \
```

```
  build-essential cmake ninja-build git \
  libboost-all-dev \
  libssl-dev zlib1g-dev \
  libsqlite3-dev
```

Ubuntu/Debian (extended graphics/text stack, feature-dependent)

```
sudo apt install -y \
  libgraphicsmagick++-dev \
  libpango1.0-dev libharfbuzz-dev \
  libgl1-mesa-dev \
  libhpdf-dev
```

Fedora/RHEL-like (baseline + TLS + SQLite)

```
sudo dnf install -y \
  gcc-c++ cmake ninja-build git \
  boost-devel \
  openssl-devel zlib-devel \
  sqlite-devel
```

3.1.4 Windows requirements

Compiler and IDE toolchain

On Windows, a professional Wt development environment is typically built around Visual Studio:

- **Visual Studio (MSVC toolset 14.1+)** with the "Desktop development with C++" workload
- **CMake 3.13+** (either the Visual Studio bundled CMake or standalone)

- **A supported Boost version (1.66+)**

Recommended Windows build utilities

While you can build with Visual Studio generators and MSBuild, many teams standardize on:

- **Ninja** (fast builds, consistent CI behavior)
- **Git** (source control, dependency fetching)
- **PowerShell** (automation scripts)

Dependency management on Windows (recommended practice)

On Windows, the most reliable way to avoid manual dependency compilation is to use a C++ package manager. Two widely used professional approaches are:

- **vcpkg-based workflow** (common with MSVC + CMake)
- **Conan-based workflow** (common cross-platform dependency profiles)

A typical vcpkg flow installs Boost/OpenSSL/zlib and then builds Wt against the vcpkg toolchain file.

Example (vcpkg-style dependency installation)

```
# Example dependencies often needed for production Wt builds:
# - boost
# - openssl
# - zlib
# Database client libs as needed (e.g., sqlite3, postgresql, mariadb)

vcpkg install boost openssl zlib sqlite3
```

3.1.5 Practical guidance: choose your dependency profile

For a book project and a professional learning path, it is useful to define **three dependency profiles** and stick to them consistently throughout the chapters:

Profile A: Minimal learning build

- CMake + compiler + Boost
- Suitable for compiling and running the first examples

Profile B: Production web server build

- Profile A + OpenSSL (+ zlib commonly)
- Suitable for HTTPS, secure cookies, and realistic deployment flows

Profile C: Full application stack build

- Profile B + one `Wt::Dbo` backend (SQLite first is recommended for learning)
- Optional graphics/text/PDF stack only if your project requires it

3.1.6 Summary

On Linux and Windows, the stable baseline for Wt development is a modern C++ compiler, CMake (3.13+), and Boost (1.66+). From that baseline, production-grade environments typically add OpenSSL for TLS and at least one database client library if `Wt::Dbo` is used. Optional graphics, text shaping, OpenGL, PDF, and FastCGI dependencies should be enabled only when the application scope requires them, to keep the build reproducible and the learning path clean.

3.2 Installing Wt (packages vs source)

3.2.1 Two professional installation strategies

There are two correct and widely used ways to install Wt for development and deployment:

1. **Install from distribution packages** (system package manager)
2. **Build from source** (CMake-based, with explicit configuration)

Both approaches are valid. The correct choice depends on whether your priority is *speed and simplicity* (packages) or *control and reproducibility* (source).

3.2.2 Option A: Install from packages (fast, simple, system-integrated)

What you gain

Installing Wt from packages is the fastest way to get started, especially on Linux:

- the distribution resolves dependencies automatically (Boost, OpenSSL, zlib, etc.),
- headers and libraries are installed in standard system locations,
- CMake can often find Wt without additional hints,
- security updates may be delivered through the OS update mechanism.

What you lose

The tradeoffs are important for professional projects:

- package versions may lag behind upstream,

- build options are fixed by the distribution maintainer,
- optional components may be disabled (or enabled) differently than you need,
- mixing multiple Wt versions across environments becomes harder.

Typical Linux examples

Exact package names vary by distribution. The following commands show the common pattern:

Debian/Ubuntu family

```
sudo apt update

# Common pattern: runtime + development headers (names vary by release)
sudo apt install -y \
  libwt-dev \
  cmake ninja-build \
  libboost-all-dev \
  libssl-dev zlib1g-dev
```

Fedora/RHEL family

```
sudo dnf install -y \
  wt-devel \
  cmake ninja-build \
  boost-devel \
  openssl-devel zlib-devel
```

Arch family

```
sudo pacman -S --needed \
  wt \
  cmake ninja \
  boost \
  openssl zlib
```

When packages are the right choice

Packages are usually the best choice when:

- you want to start learning quickly,
- you accept the distribution version,
- you do not require custom build flags,
- your deployment environment matches your development environment closely.

3.2.3 Option B: Build from source (control, reproducibility, production alignment)

What you gain

Building Wt from source is the professional default when you need control:

- choose the exact Wt version (for reproducible builds),
- enable/disable optional features explicitly (TLS, connectors, Dbo backends, etc.),
- align development, CI, and production environments,
- standardize your toolchain and dependency versions.

What you must manage

You assume responsibility for:

- dependency installation (Boost, OpenSSL, database client libs),
- selecting build type (Debug/Release),
- consistent installation prefixes and runtime paths,
- ensuring the correct Wt is found by your application CMake.

Recommended source-build workflow on Linux

This workflow is stable across most environments and works well for teaching and production.

Step 1: install dependencies

```
# Example dependency set (names vary by distribution):
# - compiler toolchain
# - CMake + Ninja
# - Boost
# - OpenSSL + zlib
# - Optional DB libs (e.g., sqlite3) if you use Wt::Dbo

# Debian/Ubuntu example:
sudo apt update
sudo apt install -y \
  build-essential cmake ninja-build git \
  libboost-all-dev \
  libssl-dev zlib1g-dev \
  libsqlite3-dev
```

Step 2: clone and build

```
git clone <WT_SOURCE_REPOSITORY>
cd wt

cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX=/usr/local

cmake --build build
sudo cmake --install build
```

Step 3: verify installation

```
# Typical sanity checks:
# - headers present under include/
# - libs present under lib/ or lib64/
# - CMake package config can be found by your projects

ls /usr/local/include/Wt
ls /usr/local/lib | grep -i wt
```

Recommended source-build workflow on Windows

On Windows, building from source is also common, but dependency management must be explicit. The most reliable professional practice is to use a C++ package manager to provide Boost/OpenSSL/zlib and then build Wt against that dependency set.

vcpkg-style approach (recommended for consistency)

```
# Install dependencies (examples, choose what you need):
```

```
vcpkg install boost openssl zlib sqlite3

# Then configure Wt with CMake using the vcpkg toolchain file:
cmake -S . -B build -G "Ninja" `
  -DCMAKE_BUILD_TYPE=Release `
  -DCMAKE_TOOLCHAIN_FILE="C:\path\to\vcpkg\scripts\buildsystems\vcpkg.cmake"

cmake --build build
cmake --install build --prefix "C:\wt"
```

When building from source is the right choice

Building from source is usually the right choice when:

- you need specific build options (TLS, DB backends, feature flags),
- you must standardize builds across dev/CI/production,
- you want a controlled upgrade path,
- you plan to teach Wt with a stable, reproducible environment.

3.2.4 Choosing between packages and source: a decision matrix

A practical decision rule:

- **Use packages** for fast onboarding, learning, and quick prototypes on a standard distro image.
- **Use source builds** for production systems, long-term maintenance, controlled upgrades, and consistent teaching environments.

3.2.5 Book recommendation: what this book should assume

For a professional book that targets “production-grade server-side applications”, the most defensible default is:

- **Teach using source builds (primary path)** to ensure readers can reproduce the same result everywhere.
- **Provide package installation as a fast alternative** for readers who want immediate progress.

This dual-path approach ensures the book remains practical for beginners while still meeting production expectations.

3.3 CMake configuration for Wt projects

3.3.1 Why CMake is the professional baseline for Wt

Wt is a native C++ framework distributed as headers and compiled libraries. Professional projects therefore require a build system that can:

- locate and link external libraries reliably across platforms,
- express build types (Debug/Release) correctly,
- scale from small examples to multi-target applications and tests,
- integrate with package managers and CI pipelines.

CMake is the most common baseline for Wt projects because it supports modern target-based linkage and enables portable builds on Linux and Windows.

3.3.2 Target-based linking: the core rule

A production-grade CMake configuration should be **target-based**. The guiding principle is:

> Do not manually add include directories and libraries globally if a package provides imported targets; link to targets and let CMake propagate usage requirements.

When Wt is installed correctly with CMake package configuration files, you should prefer:

- `find_package(Wt REQUIRED ...)`
- `target_link_libraries(app PRIVATE Wt::Wt ...)`

This automatically propagates include paths, link libraries, and compilation definitions.

3.3.3 Minimal Wt application: canonical CMake structure

The following is a minimal professional `CMakeLists.txt` suitable for the first projects in this book. It uses modern CMake practices and works with a typical Wt installation that exports CMake targets.

```
cmake_minimum_required(VERSION 3.20)
project(wt_demo LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

find_package(Wt REQUIRED COMPONENTS Wt HTTP)
```

```
add_executable(wt_demo main.cpp)

target_link_libraries(wt_demo PRIVATE Wt::Wt Wt::HTTP)

# Optional: keep warnings disciplined (example set; adjust per compiler policy)
if (MSVC)
  target_compile_options(wt_demo PRIVATE /W4 /permissive-)
else()
  target_compile_options(wt_demo PRIVATE -Wall -Wextra -Wpedantic)
endif()
```

This setup:

- selects C++20,
- locates Wt via `find_package`,
- links the correct Wt components by targets,
- stays portable across compilers.

3.3.4 Build directory discipline (out-of-source builds)

A professional workflow uses out-of-source builds:

- keeps sources clean,
- supports multiple configurations (Debug/Release) side-by-side,
- reduces accidental build artifacts committed to source control.

Recommended commands:

```
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Debug
cmake --build build
```

For Release builds:

```
cmake -S . -B build-release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build-release
```

3.3.5 When CMake cannot find Wt

CMake can only find Wt automatically if the installation provides a package configuration and the prefix is visible to CMake. In professional environments, you solve this by providing one of the following:

Using `CMAKE_PREFIX_PATH`

If Wt was installed to a non-standard prefix (for example `/opt/wt` or `C:\wt`), you can help CMake:

```
cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_PREFIX_PATH=/opt/wt
```

On Windows PowerShell:

```
cmake -S . -B build -G "Ninja" `
  -DCMAKE_BUILD_TYPE=Release `
  -DCMAKE_PREFIX_PATH="C:\wt"
```

Using a toolchain file (package manager integration)

When using a package manager (for example vcpkg), the most robust method is a toolchain file:

```
cmake -S . -B build -G "Ninja" `
  -DCMAKE_BUILD_TYPE=Release `
  -DCMAKE_TOOLCHAIN_FILE="C:\path\to\vcpkg\scripts\buildsystems\vcpkg.cmake"
```

This ensures all dependencies are resolved consistently from the same ecosystem.

3.3.6 Selecting the right Wt components

Wt is organized into components. In most applications you will use at least:

- `Wt` (core widgets, application model)
- `HTTP` (server support)

In addition, you may enable:

- `Dbo` (ORM/data layer) and backend libraries depending on database choices,
- optional graphics/text/PDF components depending on your build configuration.

Your `find_package` line should request the components you truly use, and your `target_link_libraries` should link those targets explicitly. This makes the dependency graph clear and stable.

3.3.7 RPATH and runtime library discovery (Linux)

When installing Wt into non-standard prefixes on Linux, runtime loader paths may require attention. The professional approach is:

- prefer installing to a standard location (`/usr` or `/usr/local`) when possible,
- or configure `LD_LIBRARY_PATH` for development,
- or set RPATH policies for packaged deployment.

A typical development approach (not for production packaging) is:

```
export LD_LIBRARY_PATH=/opt/wt/lib:$LD_LIBRARY_PATH
```

Production deployment is typically handled via standard library paths, container images, or packaging systems, and should be treated as an operational topic rather than a build hack.

3.3.8 A scalable CMake layout for real projects

As the book moves into the live project, you will benefit from a structured build layout. A common professional pattern:

- an `app` target (the server executable),
- one or more `lib` targets (domain/services/data layer),
- an optional `tests` target set,
- a dedicated `cmake/` directory for modules and toolchain helpers.

Example high-level structure:

```
project-root/
  CMakeLists.txt
  src/
    app/
      main.cpp
```

```
    core/
      services/
      data/
      domain/
  cmake/
  tests/
```

This reduces coupling between UI, services, and persistence and keeps build logic maintainable.

3.3.9 A clean multi-target example

The following is a compact but professional pattern that you can grow safely:

```
cmake_minimum_required(VERSION 3.20)
project(wt_app LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

find_package(Wt REQUIRED COMPONENTS Wt HTTP)

add_library(app_core STATIC
  src/core/services/user_service.cpp
  src/core/data/user_repo.cpp
)

target_include_directories(app_core PUBLIC src)
target_compile_features(app_core PUBLIC cxx_std_20)
```

```
add_executable(wt_app src/app/main.cpp)
target_link_libraries(wt_app PRIVATE app_core Wt::Wt Wt::HTTP)
```

This separates core logic from the server entry point, which is essential for long-term maintainability and testability.

3.3.10 Summary

A production-grade Wt build with CMake should be target-based, component-aware, and reproducible. Use `find_package(Wt REQUIRED COMPONENTS ...)` and link imported targets such as `Wt::Wt` and `Wt::HTTP`. Keep builds out-of-source, use `CMAKE_PREFIX_PATH` or a toolchain file when needed, and adopt a scalable multi-target structure early so that your project can evolve from "first example" into a maintainable production system.

3.4 Establishing a clean project structure from day one

3.4.1 Why structure matters more in Wt than in toy web stacks

Wt applications are not "a few request handlers". They are long-running C++ services with:

- per-session application instances and widget trees,
- event-driven UI code that can grow quickly,
- shared resources (configuration, logging, database pools),
- domain logic that must remain testable and stable,
- production deployment requirements (service management, reverse proxy, TLS).

If structure is postponed, the project tends to collapse into a single file where widgets, database calls, validation rules, and deployment assumptions are mixed. That is the fastest way to produce an unmaintainable system.
A clean structure from day one is not bureaucracy. It is a defensive engineering decision that protects maintainability, correctness, and the ability to scale the codebase and the team.

3.4.2 Design goals for the project layout

A professional Wt project structure should satisfy these goals:

1. **Separate UI wiring from domain rules.**
2. **Isolate persistence and database concerns.**
3. **Keep `WApplication` thin and focused.**
4. **Make unit testing possible without a running server.**
5. **Make configuration and operational behavior explicit.**
6. **Support growth to multiple executables and libraries.**

3.4.3 Layers that should exist early

The cleanest mental model for a Wt production system is a layered architecture. The exact naming can vary, but the responsibilities must not.

UI layer (views/widgets)

- Creates widgets and layouts
- Binds signals (events) to handlers

- Displays validation errors and results
- Does **not** implement business rules
- Does **not** directly access the database

Controller layer (orchestrates UI to services)

- Receives UI events as intentions (e.g., “CreateUser”, “ApproveCase”)
- Calls services and interprets results
- Performs mapping between UI input and domain commands
- Keeps the UI thin by centralizing workflow orchestration

Service layer (domain rules)

- Implements business logic and invariants
- Defines commands and use-cases
- Performs authorization checks (based on session context)
- Uses repositories/data access through interfaces

Data layer (persistence)

- Owns database connections, transactions, repositories
- Maps domain objects to storage representation
- Encapsulates SQL or ORM details (`Wt::Dbo` if used)
- Provides stable interfaces to services

Infrastructure layer (cross-cutting concerns)

- Configuration loading
- Logging setup
- Clock/time abstraction (useful for tests)
- Dependency wiring and service container patterns (if used)

3.4.4 A production-friendly directory layout

The following layout is intentionally conservative and scalable. It supports:

- one main server executable,
- a reusable core library,
- a clean boundary between UI and domain logic,
- test targets later without restructuring.

```
wt-project/
  CMakeLists.txt
  cmake/
  config/
    wt_config.xml
    app.production.yaml
    app.development.yaml
  assets/
    css/
    images/
  src/
```

```
  app/
    main.cpp
    server/
      app_entry.cpp
      app_entry.hpp
      session_context.hpp
  ui/
    pages/
    widgets/
    theme/
  controllers/
  domain/
    models/
    commands/
    errors/
  services/
  data/
    dbo/
    repositories/
    migrations/
  infra/
    config/
    logging/
    util/
tests/
scripts/
docs/
```

This structure makes it obvious where new code belongs and prevents accidental coupling.

3.4.5 Core principle: keep `WApplication` minimal

In Wt, every session constructs a `WApplication`. If you fill `WApplication` with business logic, your architecture will degrade quickly.
A disciplined `WApplication` should:

- initialize the root layout and navigation,
- create the initial page/view,
- hold session context (identity, role, tenant),
- provide access to shared application services through safe references,
- remain free of SQL and business rules.

3.4.6 A clean separation example: UI event to service call

The following example demonstrates the boundary: UI collects input and calls a controller/service, but the domain rules stay out of UI widgets.

Domain command and validation result

```
#include <string>
#include <vector>

struct CreateUserCommand {
  std::string username;
  std::string email;
};

struct OperationResult {
```

```
  bool ok{false};
  std::vector<std::string> errors;

  static OperationResult success() { return {true, {}}; }
  static OperationResult failure(std::vector<std::string> e) { return {false,
  ↪  std::move(e)}; }
};
```

Service implements rules (not UI)

```
class UserService {
public:
  OperationResult createUser(const CreateUserCommand& cmd) {
    std::vector<std::string> errors;

    if (cmd.username.size() < 3)
    ↪  errors.push_back("Username must be at least 3 characters.");
    if (cmd.email.find('@') == std::string::npos)
    ↪  errors.push_back("Email must contain '@'.");

    if (!errors.empty()) return OperationResult::failure(std::move(errors));

    // Persist using repository here (not shown).
    return OperationResult::success();
  }
};
```

UI calls service and displays result

```
#include <Wt/WContainerWidget.h>
```

```
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class CreateUserWidget final : public Wt::WContainerWidget {
public:
  explicit CreateUserWidget(UserService& service)
    : service_(service)
  {
    username_ = addWidget(std::make_unique<Wt::WLineEdit>());
    email_    = addWidget(std::make_unique<Wt::WLineEdit>());
    status_   = addWidget(std::make_unique<Wt::WText>());

    auto btn = addWidget(std::make_unique<Wt::WPushButton>("Create"));
    btn->clicked().connect([this] {
      CreateUserCommand cmd{ username_->text().toUTF8(), email_->text().toUTF8() };
      auto r = service_.createUser(cmd);

      if (r.ok) {
        status_->setText("Created.");
      } else {
        status_->setText("Error: " + joinErrors(r.errors));
      }
    });
  }

private:
  static std::string joinErrors(const std::vector<std::string>& e) {
    std::string out;
```

```
    for (const auto& s : e) { if (!out.empty()) out += " "; out += s; }
    return out;
  }

  UserService& service_;
  Wt::WLineEdit* username_{nullptr};
  Wt::WLineEdit* email_{nullptr};
  Wt::WText* status_{nullptr};
};
```

The UI remains responsible for presentation and event wiring. The service remains responsible for rules. This separation is the seed of maintainability.

3.4.7 CMake structure: libraries first, executable second

The build system should enforce structure. A clean pattern is:

- build most logic as a library target (`app_core`),
- build the Wt server executable as a thin target linking the library.

```
add_library(app_core STATIC
  src/services/user_service.cpp
  src/data/user_repo.cpp
  src/infra/config/app_config.cpp
)

target_include_directories(app_core PUBLIC src)
target_compile_features(app_core PUBLIC cxx_std_20)

add_executable(wt_server
```

```
  src/app/main.cpp
)

find_package(Wt REQUIRED COMPONENTS Wt HTTP)
target_link_libraries(wt_server PRIVATE app_core Wt::Wt Wt::HTTP)
```

This layout prevents accidental inclusion of UI dependencies in domain code and prepares the project for testing.

3.4.8 Configuration and deployment assets: treat them as first-class

Professional Wt deployments typically include:

- a Wt configuration file (`wt_config.xml` or equivalent),
- application configuration (dev/prod),
- static assets (CSS, images),
- service scripts for Linux deployments.

Keeping these in dedicated directories (`config/`, `assets/`, `scripts/`) makes deployment reproducible and avoids hidden assumptions.

3.4.9 Testing readiness from the beginning

Even if the project starts without tests, the structure must enable them. The key rule:

> Domain and service code must compile and run without any dependency on `Wt::WApplication` or widget headers.

If services depend on UI headers, tests become difficult and architectural boundaries collapse.

3.4.10 A disciplined growth path

A clean structure should support the project as it evolves:

- Chapter 3–4: one server executable + simple UI pages
- Later: add controllers/services/data layer
- Later: add authentication, sessions, roles, and audit logging
- Later: add CI, tests, packaging, deployment scripts

The structure above supports this growth without disruptive rewrites.

3.4.11 Summary

A clean Wt project structure from day one is a strategic choice that reduces long-term risk. The recommended approach is to separate UI widgets from controllers, services, and data access; keep `WApplication` minimal; define a scalable directory layout; and enforce boundaries through a library-first CMake design. This architecture aligns with how Wt works internally (stateful per-session applications) and prepares the codebase for professional deployment, maintenance, and teaching.

Your First Wt Application (Hello Wt)

4.1 Analyzing a minimal example

4.1.1 Purpose of a minimal example

Before building a complete application, it is essential to understand the simplest working Wt program in depth. A minimal example serves as a foundation: it reveals how Wt initializes, how the server entry point is registered, how the application instance is created per session, and how the widget tree produces output for the browser.
This section deconstructs a minimal Wt application to highlight the fundamental building blocks.

4.1.2 The code: a minimal Wt application

The following code is the canonical minimal example for starting a Wt application. It compiles into an executable that starts a server, listens for HTTP connections, and returns a simple web page containing a text widget.

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <Wt/WServer.h>
```

```
#include <memory>

class HelloApplication : public Wt::WApplication {
public:
  HelloApplication(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Hello Wt Application");
    root()->addWidget(std::make_unique<Wt::WText>("Hello from Wt"));
  }
};

int main(int argc, char** argv)
{
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);

  server.addEntryPoint(Wt::EntryPointType::Application,
    [](const Wt::WEnvironment& env) {
      return std::make_unique<HelloApplication>(env);
    });

  return server.run();
}
```

We now analyze each part.

4.1.3 Include directives and core classes

The first lines bring in the Wt classes needed for the application:

```
#include <Wt/WApplication.h>
```

```
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <Wt/WServer.h>
```

- `Wt::WApplication` is the base class for all sessions.
- `Wt::WContainerWidget` provides a container for other widgets.
- `Wt::WText` is a simple widget that renders text in the browser.
- `Wt::WServer` is the server engine that accepts connections and dispatches sessions.

4.1.4 Defining the application class

The class `HelloApplication` inherits from `Wt::WApplication`. This inheritance is crucial: it enables the Wt runtime to manage lifecycle, event dispatch, and update synchronization for each user session.

```
class HelloApplication : public Wt::WApplication {
public:
  HelloApplication(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Hello Wt Application");
    root()->addWidget(std::make_unique<Wt::WText>("Hello from Wt"));
  }
};
```

Key points:

- The constructor receives a `Wt::WEnvironment` reference, which encapsulates request metadata and session context.

- `setTitle()` sets the browser window title.
- `root()` returns the root container widget for this session.
- `addWidget()` appends a child widget to the container.

Because this class is instantiated for each session, widget state is session-specific and isolated.

4.1.5 The server entry point

The `main()` function is responsible for starting the server, registering the application entry point, and running the event loop.

```
int main(int argc, char** argv)
{
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);
```

Here:

- `Wt::WServer` constructs a server instance.
- The macro `WTHTTP_CONFIGURATION` is a built-in flag that tells Wt to use default HTTP configuration; in professional systems you will replace this with a configuration file.

4.1.6 Registering the application entry point

The next part of `main()` registers how sessions are created from incoming requests:

```
  server.addEntryPoint(Wt::EntryPointType::Application,
    [](const Wt::WEnvironment& env) {
      return std::make_unique<HelloApplication>(env);
    });
```

Understanding this line is central:

- `addEntryPoint` tells the server which function to call when a new session arrives.
- The first argument `Wt::EntryPointType::Application` indicates that this entry point is a standard application.
- The second argument is a factory callback that constructs a new `HelloApplication` instance, capturing the environment.

Because the callback returns a `std::unique_ptr`, ownership of the session object is managed safely by the runtime.

4.1.7 Running the server

The last line launches the servers event loop:

```
  return server.run();
}
```

This call:

- binds to the configured port,
- starts listening for incoming connections,
- runs the event loop that handles HTTP or WebSocket messages,
- dispatches incoming requests to create `WApplication` instances,
- returns when the server is stopped.

4.1.8 What happens at runtime

At runtime this sequence occurs:

1. The executable starts and configures the server.
2. A network listener is opened.
3. A browser requests the application URL.
4. Wt uses the entry point callback to create a new `HelloApplication` for that session.
5. The `HelloApplication` constructor builds the widget tree with a text widget.
6. The server sends the rendered UI back to the browser.
7. User interactions generate events that the server dispatches back to the session.

Even though this example only displays static text, the mechanism supports full interaction with stateful widgets.

4.1.9 Extending minimal examples

Starting from this minimal foundation, the book will demonstrate:

- adding form inputs,
- wiring event handlers,
- connecting to services and domain logic,
- integrating persistence and configuration,
- adding sessions and authorization context,

- deploying behind reverse proxies with HTTPS.

This example should remain small and focused, isolating the core mechanics of session initiation, widget composition, and event loop participation.

4.1.10 A professional note on configuration

In real systems, you replace the built-in configuration with a proper server configuration file. For example, the server constructor might be:

```
Wt::WServer server(argc, argv, "/etc/wt/wt_config.xml");
```

This makes the networking and TLS configuration explicit and manageable.

4.1.11 Summary

This minimal example reveals the following essential points:

- Wt applications are long-running server processes.
- Each session results in a new `WApplication` instance.
- The widget tree defines UI state and interaction points.
- The servers event loop underpins request handling.
- Proper separation and layering emerge from the minimal foundation.

Understanding this simple program prepares the reader to build interactive, stateful, and production-ready Wt applications in subsequent chapters.

4.2 Understanding `main()` and `WServer`

4.2.1 Why `main()` is part of the framework contract

In Wt, your program is not a set of functions invoked by an external web server runtime. Your program *is* the server-side application process. This changes what `main()` represents:

- In many web stacks, `main()` is hidden (managed runtimes, framework CLIs, serverless handlers).
- In Wt, `main()` is the explicit, authoritative entry point that defines how the service is hosted.

Therefore, `main()` is where you decide:

- which network server runtime is used,
- which entry points exist (application, resources),
- how configuration is loaded,
- how the process behaves operationally (logging, shutdown, deployment expectations).

For a production-grade Wt system, treating `main()` as an afterthought is a common failure. It should be small, clear, and intentionally structured.

4.2.2 What `Wt::WServer` is

`Wt::WServer` is the host runtime that turns your executable into a web service. It is responsible for:

- reading and applying server configuration (ports, connectors, TLS settings),

- initializing the HTTP server infrastructure,
- accepting incoming connections,
- mapping requests to entry points,
- creating a `WApplication` instance per session (through your factory),
- running the event loop until termination.

In other words, `WServer` is the boundary between the operating system (sockets, threads, signals) and your application-level logic (widgets, events, session state).

4.2.3 The canonical `main()` structure

A professional `main()` in Wt tends to follow this sequence:

1. Construct `WServer` with configuration sources
2. Register application entry points
3. Optionally register extra resources (static or dynamic)
4. Start the server event loop via `run()`
5. Return an appropriate exit code

A minimal but correct template looks like:

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WServer.h>
#include <memory>
```

```
class App final : public Wt::WApplication {
public:
  explicit App(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Hello Wt");
  }
};

int main(int argc, char** argv)
{
  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);

  server.addEntryPoint(Wt::EntryPointType::Application,
    [](const Wt::WEnvironment& env) {
      return std::make_unique<App>(env);
    });

  return server.run();
}
```

This is deliberately simple: it shows the contract between `WServer` and the application factory.

4.2.4 Understanding the `WServer` constructor

In most beginner examples, `WServer` is constructed using:

```
Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);
```

Conceptually, this constructor does three things:

- consumes command-line arguments that can influence server configuration,
- loads a configuration source (often a default or a file path),
- prepares the server runtime so it can be started by `run()`.

In production systems, configuration must be explicit and environment-specific. It is common to pass a configuration file path rather than rely on defaults:

```
Wt::WServer server(argc, argv, "/etc/wt/wt_config.xml");
```

This makes the operational behavior reproducible and avoids surprises between development and deployment.

4.2.5 Entry points: what you are really registering

`WServer::addEntryPoint` registers a mapping from incoming requests to a concrete behavior. The most common entry point is the application itself:

```
server.addEntryPoint(Wt::EntryPointType::Application,
  [](const Wt::WEnvironment& env) {
    return std::make_unique<App>(env);
  });
```

Key internal meaning:

- The server will call this factory when it needs to create a new session.
- The returned object becomes the authoritative per-session application instance.
- Ownership is transferred to the Wt runtime via `std::unique_ptr`.

This line is the origin of the "one `WApplication` per session" model.

4.2.6 What happens when `run()` is called

`server.run()` is the boundary where configuration becomes behavior. When called, `run()` typically:

- validates the configuration and initializes connectors,
- binds to network addresses/ports,
- initializes the internal request/event dispatch machinery,
- enters the server event loop,
- blocks until the server is stopped or an error forces termination,
- returns an exit code that reflects success or failure.

From an application perspective, `run()` is "start serving requests and do not return until shutdown".

4.2.7 Request routing and session lookup (conceptual)

When an incoming request arrives, `WServer` is responsible for:

1. selecting the correct entry point (application vs resource),
2. determining whether the request belongs to an existing session,
3. if it is a new session, invoking the application factory,
4. dispatching the request or event into the correct session context,
5. returning the response or incremental UI updates.

This is why `WServer` is a host runtime, not a convenience object.

4.2.8 Threading and concurrency at the `WServer` level

The concurrency model is operationally important. A single process can handle many sessions, and it may use multiple threads depending on configuration. From a project perspective:

- per-session state lives inside each `WApplication` and is isolated by design,
- shared process-wide resources (pools, caches, loggers) must be thread-safe,
- long-running operations must not block the request handling pipeline.

A production-grade `main()` typically wires shared services (configuration, logging, data pools) before `run()` starts, so sessions can use them safely.

4.2.9 A production pattern: wiring shared services into sessions

A professional structure separates process-wide services from session state. A simple pattern is to create an immutable services bundle in `main()`, and capture it in the session factory.

```
struct AppServices {
  // Example placeholders: config, logger, DB pool, etc.
};

class App final : public Wt::WApplication {
public:
  App(const Wt::WEnvironment& env, const AppServices& services)
    : Wt::WApplication(env), services_(services)
  {}

private:
  const AppServices& services_;
```

```
};

int main(int argc, char** argv)
{
  AppServices services;

  Wt::WServer server(argc, argv, WTHTTP_CONFIGURATION);
  server.addEntryPoint(Wt::EntryPointType::Application,
    [&services](const Wt::WEnvironment& env) {
      return std::make_unique<App>(env, services);
    });

  return server.run();
}
```

This clarifies what is global and what is per-session, and it prevents accidental creation of heavy resources inside each session.

4.2.10 Common mistakes and the professional correction

Mistake: heavy work inside the application constructor

If you open database connections, load huge datasets, or build expensive caches in the `WApplication` constructor, the cost will repeat per session.
Correction: create shared resources once at process startup (in `main()`) and access them through safe interfaces.

Mistake: storing global mutable state in singletons

Global mutable state becomes a concurrency hazard.

Correction: prefer explicit dependency injection via captured references, and enforce thread safety in shared components.

Mistake: ignoring operational configuration

Relying on default configuration makes deployments fragile.
Correction: use explicit configuration files and environment-specific configuration management from the beginning.

4.2.11 Summary

In Wt, `main()` is the explicit hosting definition of your server-side application, and `Wt::WServer` is the runtime that provides network listening, request dispatch, session creation, and the event loop. Understanding this relationship is essential because it shapes architecture and operations: shared services belong to the process (wired in `main()`), user state belongs to sessions (inside `WApplication`), and the server runs as a managed long-lived service that must be configured, monitored, and deployed with production discipline.

4.3 Entry points explained

4.3.1 What an entry point represents

In Wt, an *entry point* defines how incoming HTTP requests are mapped to executable behavior in your application. Because Wt applications run as server processes that host multiple sessions, entry points are the mechanism by which the server knows how to construct application logic for:

- interactive sessions,
- static or dynamic resources,

- service endpoints.

In the simplest terms, an entry point is a factory function that the server invokes to produce an object representing a handling context for a given request.

4.3.2 Why entry points matter for modular design

Entry points separate:

- the server runtime concerns (network, threads, event loop), from
- the application logic (UI, business behavior).

They are central to how Wt maintains a clear separation of concerns:

- The server invokes an entry point to build a session or resource handler.
- Your code implements application-specific behavior in the object returned by the entry point.

This separation enables multiple types of entry points in one server binary and supports extensibility and modular deployments.

4.3.3 Types of entry points

Wt defines several kinds of entry points, but the most common are:

- `Wt::EntryPointType::Application` — standard interactive session entry point.
- `Wt::EntryPointType::Resources` — handler for static or dynamic resources.

Other types may exist depending on the version of the framework or extensions, but these two are the primary building blocks for most realistic systems.

4.3.4 Standard application entry point

The application entry point is used for interactive, stateful sessions. It tells the server how to construct a `WApplication` instance for each new browser session.
The canonical registration syntax is:

```
server.addEntryPoint(
  Wt::EntryPointType::Application,
  [](const Wt::WEnvironment& env) {
    return std::make_unique<MyApplication>(env);
  });
```

Here:

- `Wt::EntryPointType::Application` identifies the type.
- The lambda is the factory that constructs the application instance.
- The `WEnvironment` passed to the factory contains request metadata and session context.

The server calls this factory whenever a new session is initiated, ensuring that each session receives a fresh application instance.

4.3.5 How entry points support multiple behaviors

A Wt server can support more than one entry point. This enables a single executable to handle:

- the main interactive application,
- auxiliary REST or resource endpoints,
- diagnostic or administrative routes.

For example, a resource entry point might be registered as:

```
server.addEntryPoint(
  Wt::EntryPointType::Resources,
  [](const Wt::WEnvironment& env) {
    return std::make_unique<ResourceHandler>(env);
  });
```

A resource handler may generate images, serve files, provide API responses, or stream data.

4.3.6 How the server routes requests to entry points

Internally, the server uses URL patterns and session context to determine which entry point should handle a request. The decision can be based on:

- path segments,
- query parameters,
- cookies or headers used for session identification.

In practice, Wt applications normally attach the application entry point at the root URL, and resource entry points under specific prefixes. The configuration file or command-line options can modify default mappings.

4.3.7 Factory semantics: why unique_ptr matters

The use of `std::unique_ptr` in entry point factories reflects a clear ownership model:

- The factory returns ownership of the created object to the Wt server.
- The server is then responsible for lifetime management of that object.

- A new pointer is produced for each independent session or resource request context.

This avoids shared ownership issues and leverages deterministic cleanup when an objects lifetime ends.

4.3.8 Example: multi-entry-point configuration

In a project where an API endpoint exists alongside the main UI, entry points might be registered like this:

```
server.addEntryPoint(
  Wt::EntryPointType::Application,
  [](const Wt::WEnvironment& env) {
    return std::make_unique<MainApp>(env);
  });

server.addEntryPoint(
  Wt::EntryPointType::Resources,
  [](const Wt::WEnvironment& env) {
    return std::make_unique<ApiHandler>(env);
  });
```

Here:

- `MainApp` is the interactive, stateful session application.
- `ApiHandler` handles non-session resource requests, such as REST responses or dynamic assets.

This pattern increases modularity: UI logic is isolated from service logic, even within the same deployment artifact.

4.3.9 Entry points and session lifetime

For the application entry point:

- The entry point factory is invoked once per session.
- The returned `WApplication` instance lives for the duration of the session.
- When the session expires or ends, the framework destroys that instance.

For resource or API entry points:

- Instances may be short-lived and tied to single request contexts.
- Lifecycle rules depend on the type and implementation of the handler.

Understanding these lifetimes helps prevent resource leaks and manage concurrency boundaries responsibly.

4.3.10 Entry points and application configuration

In professional settings, entry point configuration is often complemented by explicit configuration files (for example, `wt_config.xml`). These files can define:

- listener ports,
- TLS certificates,
- maximum concurrency settings,
- virtual hosts or listener prefixes,
- resource mappings.

While the factory registration is in code, the deployment configuration remains external, allowing environment-specific tuning without recompiling.

4.3.11 Design patterns for entry point factories

In a mature application, the entry point factory often performs dependency injection. Instead of creating the application instance directly, it captures references to shared services such as:

- configuration providers,
- logging components,
- database connection pools,
- domain service containers.

A common professional pattern is:

```
struct AppServices {
  // Shared services, config, resources
};

AppServices services;

server.addEntryPoint(
  Wt::EntryPointType::Application,
  [&services](const Wt::WEnvironment& env) {
    return std::make_unique<MyApp>(env, services);
  });
```

This pattern makes cross-cutting dependencies explicit and avoids ad-hoc global state.

4.3.12 Summary

Entry points in Wt are explicit mappings from a server runtime to application behavior. They form the contract between incoming requests and the factory functions that generate session or

resource-handling objects. By registering entry points responsibly, and by separating application logic (UI sessions) from resource handlers (API, static content), you build modular and maintainable systems. Entry points should be designed with lifetimes, dependency injection, and deployment configuration in mind, setting a solid foundation for scalable Wt applications.

4.4 Running the application locally

4.4.1 What “running locally” means for Wt

A Wt application is not deployed to an application server as a script. It is built into a native executable that runs as a local server process. Running locally therefore means:

- you start the compiled executable on your machine,
- it binds to a TCP port and listens for HTTP (and optionally WebSocket) connections,
- you open a browser and connect to `http://localhost:<port>/`,
- the server creates a session and returns the rendered page.

The local run workflow is the same hosting model you will use in production, except that production adds reverse proxies, TLS, service supervision, and hardened configuration.

4.4.2 Prerequisites before the first run

To run the application locally, you must have:

- a successfully built executable (Debug or Release),
- access to the Wt runtime libraries on your system library path,

- a Wt configuration source (defaults or a config file),
- a free TCP port (commonly `8080` for local testing).

If you built Wt from source into a non-standard prefix, ensure the runtime loader can find the Wt shared libraries (Linux) or DLLs (Windows).

4.4.3 Local run on Linux

Build with CMake (recommended out-of-source)

From your project root:

```
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Debug
cmake --build build
```

Then run the executable:

```
./build/wt_demo
```

If the loader cannot find Wt shared libraries

If Wt is installed into a non-standard prefix such as `/opt/wt`, the dynamic loader may fail. A development-only workaround is:

```
export LD_LIBRARY_PATH=/opt/wt/lib:$LD_LIBRARY_PATH
./build/wt_demo
```

A more production-aligned solution is to install Wt into standard paths or package the runtime properly. For local learning, `LD_LIBRARY_PATH` is acceptable.

Confirm the server is listening

If the application starts correctly, it will typically print startup information to the console (port, configuration, and connector details). To confirm the process is listening, you may use:

```
ss -ltnp | grep 8080
```

Then open your browser:

```
http://127.0.0.1:8080/
```

4.4.4 Local run on Windows

Build with CMake + Ninja

From a Developer PowerShell or a terminal with MSVC environment loaded:

```
cmake -S . -B build -G "Ninja" -DCMAKE_BUILD_TYPE=Debug
cmake --build build
```

Run the executable:

```
.\build\wt_demo.exe
```

If DLLs are missing

On Windows, a common first-run failure is missing DLLs (Wt, Boost, OpenSSL). The correct professional solutions are:

- install dependencies via a package manager and ensure the runtime is discoverable,
- or copy required DLLs next to the executable for local development,
- or configure the PATH to include the dependency `bin` directories.

For local learning, the simplest approach is ensuring your dependency manager (such as vcpkg) provides a consistent build and runtime layout.

Open the browser

Once running, open:

```
http://127.0.0.1:8080/
```

If your configuration chooses a different port, use that value.

4.4.5 Controlling the local server port and configuration

Wt can be configured by a configuration file (commonly `wt_config.xml` in many setups) or by command-line parameters depending on how the server is initialized. In professional usage, prefer an explicit configuration file so runs are reproducible.
A typical pattern is:

```
Wt::WServer server(argc, argv, "config/wt_config.xml");
```

Then you run your executable normally, and the server reads port/connectors and other settings from the file.
For local development, you often maintain two configs:

- `config/wt_config.dev.xml`
- `config/wt_config.prod.xml`

The executable remains the same; only configuration changes.

4.4.6 How to interpret the first run

When you open the browser:

1. The browser requests the root URL.

2. Wt accepts the connection.
3. A new session is created.
4. Your registered application entry point factory constructs `WApplication`.
5. The constructor builds the widget tree.
6. Wt renders the initial page and returns it.

If your “Hello Wt” example shows a simple text, that confirms:

- server runtime is working,
- entry point registration is correct,
- session creation is functional,
- rendering pipeline is working.

4.4.7 Troubleshooting the most common local issues

Port already in use

Symptoms:

- the server fails to start or reports bind failure.

Fix:

- choose another port in the configuration,
- stop the process using the current port.

Linux helper:

```
sudo lsof -i :8080
```

Browser opens but page is blank or not updating

Likely causes include:

- missing static resources (wrong working directory, missing assets),
- proxy or firewall interference (less common locally),
- WebSocket upgrade not working (fallback should still work, but may affect interactivity).

Fix:

- confirm the process console logs show requests arriving,
- confirm the working directory and resource paths,
- disable strict proxy settings for local host.

Runtime library not found

Linux symptoms:

- error about missing shared objects.

Windows symptoms:

- “DLL not found” or application fails to start.

Fix:

- install Wt and dependencies into standard locations,
- or set runtime paths (`LD_LIBRARY_PATH` / `PATH`) for development,
- or package DLLs next to the executable for local runs.

4.4.8 Recommended local workflow for this book

To keep the learning process clean and reproducible, this book assumes:

- out-of-source builds (separate build directory),
- explicit configuration under a `config/` directory,
- a consistent default local port (`8080`),
- running directly from the terminal to see logs.

This approach aligns local development with production discipline.

4.4.9 Summary

Running a Wt application locally means running a native server executable on your machine, binding to a port, and connecting through a browser to `localhost`. The key professional concerns are correct build output, correct runtime dependency discovery, explicit configuration, and clear diagnostics via console logs. Once the "Hello Wt" example renders successfully, you have validated the full path: build system, server startup, session creation, widget rendering, and client connection.

4.5 First widget and root container

4.5.1 What the root container represents

In Wt, every `WApplication` instance owns a root container that represents the top-level UI element for a single user session. This container is the entry point into the widget hierarchy and serves a role similar to the document body in classic web development or the main window in desktop UI frameworks.

All visual content, interaction logic, and layout composition ultimately descend from this root container. Understanding it is essential because Wt applications are stateful, session-bound, and widget-driven rather than request-driven.

4.5.2 Accessing the root container

The root container is accessed via the `root()` member function of `WApplication`. This function returns a pointer to a container widget that exists for the lifetime of the session.

```
root()->addWidget(std::make_unique<Wt::WText>("Hello from Wt"));
```

This single line already expresses several key concepts:

- `root()` provides access to the top-level container.
- `addWidget` attaches a child widget to the UI tree.
- Ownership of the widget is transferred to the container.

4.5.3 Widget ownership and lifetime

Wt uses modern C++ ownership semantics for widgets. When you add a widget to a container using `std::unique_ptr`, the container becomes the owner of that widget.
This means:

- Widgets are destroyed automatically when their parent container is destroyed.
- The entire widget tree is destroyed when the session ends.
- There is no need for manual memory management for UI elements.

This ownership model is deterministic, safe, and aligned with modern C++ design principles.

4.5.4 Your first widget: `Wt::WText`

The simplest widget you can add is `Wt::WText`. It renders textual content into the browser and is typically used for labels, headings, and static messages.

```
#include <Wt/WApplication.h>
#include <Wt/WText.h>

class HelloApplication : public Wt::WApplication {
public:
  HelloApplication(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Hello Wt");

    ↪  root()->addWidget(std::make_unique<Wt::WText>("Welcome to your first Wt applicatio
  }
};
```

When a browser connects, this widget is rendered immediately as part of the sessions UI.

4.5.5 Why everything starts at the root

The root container is not optional. Every widget must ultimately have a parent, and the root is the only widget guaranteed to exist when the application is constructed.
From an architectural perspective:

- the root defines the session UI boundary,
- it isolates users from one another,
- it anchors layout, styling, and event handling.

This design ensures that Wt applications scale safely to many concurrent users.

4.5.6 Using layout containers under the root

While you can add widgets directly to the root container, professional applications almost always use layout containers to control structure and alignment.

```
#include <Wt/WVBoxLayout.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>

auto layout = root()->setLayout(std::make_unique<Wt::WVBoxLayout>());

layout->addWidget(std::make_unique<Wt::WText>("Your name:"));
layout->addWidget(std::make_unique<Wt::WLineEdit>());
layout->addWidget(std::make_unique<Wt::WPushButton>("Submit"));
```

Here:

- `setLayout` assigns a layout manager to the root container.
- The layout controls positioning and resizing.
- Widgets are still owned by the container hierarchy.

This approach scales far better than flat widget insertion.

4.5.7 Event-driven behavior from the root

Widgets added to the root can emit signals that trigger application logic. For example:

```
auto button = root()->addWidget(std::make_unique<Wt::WPushButton>("Click"));

button->clicked().connect([this] {
  root()->addWidget(std::make_unique<Wt::WText>("Button clicked"));
});
```

This demonstrates:

- signalslot style event handling,
- modification of the widget tree at runtime,
- automatic synchronization with the browser.

All changes occur within the current session only.

4.5.8 Clearing and replacing the root content

It is common to change the entire view dynamically, for example when navigating between pages or modes.

```
root()->clear();
root()->addWidget(std::make_unique<Wt::WText>("New view loaded"));
```

This pattern supports navigation systems, dashboards, and role-based interfaces.

4.5.9 Styling the root container

The root container can be styled like any other widget:

```
root()->setStyleClass("main-layout");
```

Styling is applied via CSS served by the application, enabling responsive and professional UI design without mixing styling logic into C++ code.

4.5.10 Root container and session isolation

Each browser session has its own `WApplication` instance and therefore its own root container. UI changes in one session do not affect others. This is a fundamental property that makes Wt safe for concurrent multi-user environments.

4.5.11 Summary

The root container is the foundation of every Wt user interface. It owns the widget tree, defines the session UI boundary, and provides the starting point for layout, interaction, and styling. The first widget you addoften a simple `WTextconfirms` that the session is active and rendering correctly. From this minimal beginning, complex, interactive, and production-grade interfaces are built by composing widgets under the root container using layouts, events, and clear architectural boundaries.

Part III

Professional Project Architecture

Designing a Production-Ready Architecture

5.1 Layer separation (UI / Controllers / Services / Models)

5.1.1 Why layering is non-negotiable in production Wt systems

A Wt application is a long-running C++ server process hosting many concurrent user sessions. The framework makes it easy to build a UI directly in C++ by composing widgets and connecting events. That convenience is also the main architectural trap: without strict boundaries, the project degenerates into a single layer where:

- widget event handlers contain business rules,
- business rules call SQL directly,
- validation rules are duplicated across UI code paths,
- authentication and authorization are scattered,
- changes become risky because everything depends on everything.

Layer separation prevents that collapse. It is not a pattern chosen for style; it is a correctness and maintainability strategy. The goal is to ensure that each part of the system has a single

responsibility, that dependencies flow in one direction, and that domain logic remains testable without a running web server.

5.1.2 The four-layer contract

This book uses a pragmatic four-layer contract:

1. **UI (Views/Widgets/Pages)**
2. **Controllers (Workflow Orchestration)**
3. **Services (Business Rules / Use Cases)**
4. **Models (Domain Concepts and Data Contracts)**

Each layer has a specific purpose and strict dependency rules.

5.1.3 Layer 1: UI layer (Views / Widgets / Pages)

Responsibility

The UI layer is responsible for **presentation** and **user interaction wiring**. It owns:

- widget composition (layout, pages, components),
- signal connections (clicks, form submissions, selections),
- user input collection and basic formatting,
- displaying results, errors, and navigation transitions.

Forbidden responsibilities

The UI layer must not:

- implement business rules,
- enforce authorization policy,
- open database connections or run SQL,
- contain transaction logic,
- contain infrastructure concerns (logging config, deployment config).

What the UI should call

UI should call **controllers** (preferred) or **services** (acceptable for tiny apps). In a production architecture, UI events should translate into high-level intentions, not low-level operations.

5.1.4 Layer 2: Controller layer (workflow orchestration)

Responsibility

Controllers convert UI events into application actions. They orchestrate a workflow by:

- validating input at the boundary (shape validation, required fields),
- creating domain commands and calling services,
- mapping service results to UI-friendly responses,
- coordinating multi-step flows (wizards, approvals),
- deciding navigation transitions (e.g., replace a page, show a dialog).

What controllers are not

Controllers are not:

- UI widgets,
- persistence repositories,
- a place to duplicate business rules.

Controllers must remain thin and use services for domain decisions.

5.1.5 Layer 3: Service layer (business rules and use cases)

Responsibility

Services implement the **business behavior** of the system. They define use cases such as:

- create user,
- submit request,
- approve case,
- generate report,
- revoke access,
- enforce constraints and invariants.

Services should:

- enforce domain invariants consistently,
- implement authorization checks based on caller context,
- remain independent of Wt widget types,
- depend on repositories/interfaces rather than concrete DB details.

Inputs and outputs

Services should use explicit data contracts:

- command/request objects for inputs,
- result objects for outputs (success/failure + errors + data),
- domain-specific error codes when needed.

This avoids throwing UI exceptions into the domain and keeps behavior testable.

5.1.6 Layer 4: Models (domain concepts and contracts)

Responsibility

Models define the meaning of the system. This includes:

- entities (objects with identity and lifecycle),
- value objects (immutable concepts like Email, Money),
- enums and domain states,
- commands/DTOs exchanged across layers,
- validation rules at the domain level (not UI formatting rules).

Models should not depend on UI types or Wt headers. If a model includes UI or database dependencies, the architecture is already compromised.

5.1.7 Dependency direction: the single rule that prevents chaos

A production codebase remains stable only when dependencies flow in one direction:

$$\text{UI} \rightarrow \text{Controllers} \rightarrow \text{Services} \rightarrow \text{Models}$$

Additionally:

- Services can depend on Models.
- Controllers can depend on Models and Services.
- UI can depend on everything above it, but should only call Controllers.
- Models must not depend on any other layer.

Persistence and infrastructure are typically injected into services via interfaces, keeping services stable.

5.1.8 A minimal example illustrating the separation

The following example shows a clean separation using explicit command/result objects.

Models: command and result

```
#include <string>
#include <vector>

struct CreateUserCommand {
  std::string username;
  std::string email;
};
```

```
struct OperationResult {
  bool ok{false};
  std::vector<std::string> errors;

  static OperationResult success() { return {true, {}}; }
  static OperationResult failure(std::vector<std::string> e) { return {false,
  ↪  std::move(e)}; }
};
```

Services: domain rules

```
class UserService {
public:
  OperationResult createUser(const CreateUserCommand& cmd) {
    std::vector<std::string> errors;

    if (cmd.username.size() < 3)
      errors.push_back("Username must be at least 3 characters.");

    if (cmd.email.find('@') == std::string::npos)
      errors.push_back("Email must contain '@'.");

    if (!errors.empty())
      return OperationResult::failure(std::move(errors));

    // Persist via repository (not shown). Business rules stay here.
    return OperationResult::success();
  }
};
```

Controllers: orchestration and mapping

```
class UserController {
public:
  explicit UserController(UserService& service) : service_(service) {}

  OperationResult onCreateUser(std::string username, std::string email) {
    // Boundary validation: shape, trimming, empty checks.
    if (username.empty() || email.empty())
      return OperationResult::failure({"Username and email are required."});

    CreateUserCommand cmd{std::move(username), std::move(email)};
    return service_.createUser(cmd);
  }

private:
  UserService& service_;
};
```

UI: widgets and event wiring

```
#include <Wt/WContainerWidget.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class CreateUserPage final : public Wt::WContainerWidget {
public:
  explicit CreateUserPage(UserController& controller)
    : controller_(controller)
```

```
  {
    username_ = addWidget(std::make_unique<Wt::WLineEdit>());
    email_    = addWidget(std::make_unique<Wt::WLineEdit>());
    status_   = addWidget(std::make_unique<Wt::WText>());

    auto btn = addWidget(std::make_unique<Wt::WPushButton>("Create"));
    btn->clicked().connect([this] {
      auto r = controller_.onCreateUser(
        username_->text().toUTF8(),
        email_->text().toUTF8()
      );

      if (r.ok) status_->setText("Created.");
      else      status_->setText(renderErrors(r.errors));
    });
  }

private:
  static std::string renderErrors(const std::vector<std::string>& e) {
    std::string out;
    for (const auto& s : e) { if (!out.empty()) out += " "; out += s; }
    return out;
  }

  UserController& controller_;
  Wt::WLineEdit* username_{nullptr};
  Wt::WLineEdit* email_{nullptr};
  Wt::WText* status_{nullptr};
};
```

This example demonstrates:

- UI is purely wiring and rendering.
- Controller translates UI inputs into commands.
- Service enforces business rules and returns structured results.
- Models are clean data contracts used across layers.

5.1.9 Where persistence fits (repositories as a boundary)

Most production systems add an explicit persistence boundary:

- Services depend on repository interfaces (not SQL).
- Repository implementations live in the data layer and depend on the database or ORM.

This keeps services testable and prevents data access from contaminating UI code.

5.1.10 Why this separation is especially important in Wt

Wt encourages server-side UI code that feels like desktop GUI programming. That is a strength, but it makes it easy to accidentally merge UI and business logic into one place. Layer separation prevents:

- UI-specific dependencies leaking into the domain,
- per-session widget state becoming the only source of truth,
- duplicated business rules in multiple widget event handlers,
- unsafe shared-state hacks introduced to “solve” architectural confusion.

With clear layers, Wt becomes a disciplined server-side platform rather than a monolithic widget script.

5.1.11 Summary

A production-ready Wt architecture requires strict layer separation. The UI layer builds widgets and binds events, controllers orchestrate workflows, services implement business rules and use cases, and models define domain concepts and data contracts. Dependencies must flow in one direction, keeping models pure and services independent of Wt headers. This structure preserves maintainability, testability, and correctness as the codebase and user base grow.

5.2 Why classic MVC does not directly fit Wt

5.2.1 The problem is not MVC itself, but what "classic MVC" assumes

The phrase "MVC" is overloaded. In many web frameworks, MVC is a routing-and-request pattern: a request hits a controller, the controller builds or fetches a model, and a view is rendered into HTML. That pattern assumes:

- a **stateless request/response cycle** as the primary unit of work,
- a **controller per request** (or per route),
- a **view rendering step** that produces a full HTML response (or template output),
- a **model** that is usually data shaped for the view (often a view-model or DTO),
- minimal server-held UI state between requests (state is reconstructed each time).

Wt does not follow these assumptions. Wt is a **stateful, event-driven, session-based server UI runtime**. It is closer to a desktop GUI framework hosted over HTTP/WebSockets than to a conventional stateless web framework.
This is why classic MVC does not map one-to-one to Wt, even though the underlying goals (separation of concerns) remain correct.

5.2.2 Wt's fundamental architecture: sessions and widget trees

A Wt application instance (`WApplication`) is created **per session**. That instance owns:

- a persistent widget tree,
- event handlers connected to widgets,
- session-scoped state (identity, workflow progress, UI state),
- a server-driven synchronization pipeline to update the browser incrementally.

The unit of interaction is not an HTTP request. The unit of interaction is an **event** arriving from the browser that is dispatched into the session's application instance.

5.2.3 Where classic MVC breaks

Classic MVC breaks in Wt because the boundaries do not match the runtime.

Mismatch 1: controller-per-request vs controller-per-session

In classic web MVC:

- controllers are invoked for each request,
- the controller does not persist as a long-lived interaction object,
- state is reloaded and revalidated per request.

In Wt:

- the `WApplication` and widget tree persist per session,
- events are handled by callbacks that are already connected,

- interaction context exists continuously.

Trying to force “controller-per-request” typically leads to unnatural code: you build artificial routing logic inside a stateful event loop, duplicating mechanisms that the framework already provides.

Mismatch 2: view rendering vs incremental UI updates

In classic MVC:

- a view is rendered into HTML (often a template),
- the output is usually a complete page or a server-rendered fragment.

In Wt:

- the view is the widget tree itself,
- rendering is incremental and synchronized,
- updates are often partial DOM changes rather than full page responses.

This makes the view lifecycle continuous. The view is not a one-time rendering step; it is a stateful object graph that evolves.

Mismatch 3: model meaning

In classic MVC frameworks, the “model” often becomes a view-model: a structure designed primarily to feed templates. In Wt, the UI is a persistent object graph, and a model can mean:

- domain entities and value objects (business meaning),
- session state (workflow and UI decisions),
- data sources for widgets (`WAbstractItemModel` style concepts).

These are distinct concerns. Classic MVC collapses them into a single “M”, but Wt’s design benefits from keeping them explicit.

Mismatch 4: routing is not the core primitive

Classic web MVC begins with routing:

- URL routes map to controller actions.

Wt begins with sessions and events:

- the session application is already running,
- events are dispatched to already-constructed handlers,
- navigation is often internal UI navigation, not URL-based routing.

Wt can support internal paths and URL changes, but in typical Wt applications this is not the primary organizing principle. Attempting to model everything as route handlers produces friction and tends to hide the strengths of Wt.

5.2.4 What replaces classic MVC in Wt: a better mapping

The correct approach is not to abandon separation of concerns. The correct approach is to use a separation model that aligns with a stateful, session-based runtime.
A practical mapping is:

- **View (UI)**: widgets/pages/components (persistent object graph)
- **Controller (workflow)**: session-level orchestration (intent handling)
- **Model (domain)**: entities/value objects + commands/results
- **Services**: use cases and policy enforcement (domain behavior)

This resembles “MVC” in spirit, but in structure it is closer to:

- Presentation (widgets) + Application layer (controllers/services) + Domain layer (models),

with an explicit boundary for persistence.

5.2.5 The key architectural rule: keep Wt types out of domain logic

The most important practical rule for production systems is:

> Domain services and models must not depend on Wt headers or widget types.

Wt widgets are presentation-layer objects. If they leak into services/models, you lose testability and create tight coupling.

5.2.6 A concrete example of the MVC trap

A common anti-pattern is to implement business rules directly inside a widget event handler:

```
button->clicked().connect([this] {
  // Anti-pattern: business rules and persistence inside UI code
  if (nameEdit_->text().empty()) {
    status_->setText("Name required");
    return;
  }
  db_.execute("INSERT INTO users ..."); // persistence in UI
  status_->setText("Created");
});
```

This looks convenient at first, but it couples:

- UI event wiring,
- validation rules,
- persistence calls,
- error handling policy,
- transactional correctness,

into one place. That is exactly the failure mode classic web MVC avoids, but classic MVC cannot be copied directly into Wt because the runtime boundaries are different.

5.2.7 A Wt-aligned solution: controller + service boundary

A Wt-aligned separation keeps the UI thin and delegates to a controller/service:

```
button->clicked().connect([this] {
  auto result = controller_.createUser(
    nameEdit_->text().toUTF8(),
    emailEdit_->text().toUTF8()
  );

  if (result.ok) status_->setText("Created");
  else status_->setText(result.message);
});
```

The UI remains event wiring and rendering. The controller/service layers implement rules and persistence through repositories.

5.2.8 Why this matters operationally

Wt is a long-lived process. Poor separation has operational consequences:

- memory growth due to careless session state,
- inconsistent authorization because checks are scattered,
- difficult concurrency reasoning because shared resources are accessed from UI code,
- fragile deployments because configuration is embedded into UI logic,
- inability to write unit tests without launching the server.

A Wt-aligned architecture treats the UI as one layer and isolates the business system behind services and models.

5.2.9 Summary

Classic web MVC does not directly fit Wt because classic MVC assumes stateless request/response routing, controller-per-request execution, and view rendering as a discrete step. Wt is session-based, stateful, and event-driven, where the view is a persistent widget tree and updates are incremental. The correct solution is not to abandon separation of concerns, but to adopt a Wt-aligned layering: UI widgets/pages, controllers that orchestrate session workflows, services that implement use cases and policy, and models that define domain meaning. This alignment preserves testability, maintainability, and production correctness while leveraging Wt's strengths instead of fighting its runtime model.

5.3 A scalable, project-driven architecture

5.3.1 Why “project-driven” is the only architecture that survives

A production system is not a tutorial. It is a continuously evolving product that must withstand:

- changing requirements and expanding scope,
- growth in users, data volume, and concurrency,
- multiple developers and handoffs,
- operational constraints (monitoring, logging, deployment, rollback),
- long-term maintenance under cost and time pressure.

A scalable architecture is therefore not defined by how elegant it looks on paper. It is defined by how well it supports the **project lifecycle**. “Project-driven architecture” means we choose boundaries, modules, naming, and build layout based on how the codebase will grow and how the product will be deployed and operated.
Wt increases the need for discipline because the UI is a persistent object graph per session, and the server is a long-lived C++ process. Without a scalable structure, UI logic, business rules, and persistence will merge and become impossible to evolve safely.

5.3.2 Design goals for a scalable Wt architecture

A scalable architecture for Wt should satisfy these goals from day one:

1. **Stable boundaries:** UI changes should not rewrite business rules; database changes should not rewrite widgets.

2. **One-direction dependencies:** dependencies must flow toward the domain, never outward from it.
3. **Testability:** services and domain logic must be testable without starting `WServer`.
4. **Operational clarity:** configuration, logging, and lifecycle behaviors must be explicit.
5. **Session safety:** per-session state is isolated; shared resources are thread-safe and controlled.
6. **Incremental growth:** new features add modules, not complexity to existing modules.

5.3.3 A minimal set of modules that scale

A scalable Wt project is best organized into modules (libraries) rather than a single executable containing everything. A practical module set is:

- **app** (server entry, Wt integration, composition root)
- **ui** (widgets/pages, view composition, navigation)
- **controllers** (workflow orchestration and boundary validation)
- **domain** (models, invariants, errors, value objects)
- **services** (use-cases, policies, authorization checks)
- **data** (repositories, database sessions, mapping, migrations)
- **infra** (configuration, logging, time, utilities, external adapters)

This structure is simple enough for a small project, yet powerful enough to grow.

5.3.4 The dependency rule that prevents architectural collapse

The dependency rule must be strict:

> **UI and Wt code depend on services and domain. Domain depends on nothing.**

In practice:

- `domain` has no dependency on Wt, DB libraries, or platform code.
- `services` depends on `domain` and repository interfaces.
- `data` implements repository interfaces and depends on DB/ORM.
- `ui` depends on `controllers` and view models, not on `data`.
- `app` wires everything together and starts `WServer`.

This is how you keep Wt a presentation layer rather than the center of the universe.

5.3.5 A directory layout that supports scaling

The following directory layout is designed to support long-term growth, multiple targets, tests, and deployment assets:

```
project/
  CMakeLists.txt
  cmake/
  config/
    wt_config.dev.xml
    wt_config.prod.xml
    app.dev.yaml
    app.prod.yaml
```

```
assets/
  css/
  images/
src/
  app/
    main.cpp
    bootstrap/
      composition_root.cpp
      composition_root.hpp
    session/
      session_context.hpp
  ui/
    pages/
    widgets/
    navigation/
  controllers/
  domain/
    model/
    value/
    errors/
  services/
    use_cases/
    policies/
  data/
    repositories/
    dbo/
    migrations/
  infra/
    config/
```

```
    logging/
    util/
tests/
scripts/
docs/
```

5.3.6 Composition root: the architecture hinge in Wt

In scalable systems, there must be exactly one place where everything is wired together. This is called the **composition root**. In Wt, the natural composition root is near `main()`:

- build shared services (configuration, logging, DB pools),
- register entry points,
- create per-session application instances with injected dependencies,
- start `WServer`.

A composition root prevents dependency chaos and eliminates global singletons.

5.3.7 A disciplined session model

Because Wt is stateful per session, you must define what is allowed to live in session memory. A professional rule is:

- session holds identity, role, tenant, workflow state, UI state,
- session does **not** hold large datasets or persistent domain objects,
- session holds identifiers and minimal derived values,
- persistent objects are loaded through services as needed.

This prevents memory growth and avoids stale state bugs.

5.3.8 Use cases as the unit of growth

The best scaling unit is a **use case**. Each new feature adds:

- a command (input contract),
- a result (output contract),
- a service method implementing the use case,
- a controller method orchestrating UI input to the service,
- UI widgets/pages that present the use case.

This enforces consistent growth and reduces the risk of ad-hoc feature code.

5.3.9 An example: use-case centered structure

Domain contracts

```
#include <string>
#include <vector>

struct CreateCaseCommand {
  std::string title;
  std::string description;
};

struct UseCaseResult {
  bool ok{false};
  std::vector<std::string> errors;
  std::string id;
```

```
  static UseCaseResult success(std::string newId) {
    UseCaseResult r; r.ok = true; r.id = std::move(newId); return r;
  }
  static UseCaseResult failure(std::vector<std::string> e) {
    UseCaseResult r; r.ok = false; r.errors = std::move(e); return r;
  }
};
```

Service implements the use case

```
class CaseService {
public:
  UseCaseResult createCase(const CreateCaseCommand& cmd) {
    std::vector<std::string> errors;
    if (cmd.title.size() < 5)
    ↪  errors.push_back("Title must be at least 5 characters.");
    if (!errors.empty()) return UseCaseResult::failure(std::move(errors));

    // Persist via repository and return created id (not shown).
    return UseCaseResult::success("CASE-001");
  }
};
```

Controller orchestrates input

```
class CaseController {
public:
  explicit CaseController(CaseService& service) : service_(service) {}
```

```
  UseCaseResult onCreateCase(std::string title, std::string description) {
    if (title.empty()) return UseCaseResult::failure({"Title is required."});
    CreateCaseCommand cmd{std::move(title), std::move(description)};
    return service_.createCase(cmd);
  }

private:
  CaseService& service_;
};
```

The architecture scales because each feature is added in a consistent pattern.

5.3.10 CMake: enforce modules as targets

A scalable project should enforce module boundaries at the build level by using library targets:

```
add_library(domain STATIC
  src/domain/model/case.cpp
  src/domain/value/email.cpp
)

add_library(services STATIC
  src/services/case_service.cpp
)
target_link_libraries(services PUBLIC domain)

add_library(ui STATIC
  src/ui/pages/create_case_page.cpp
)
target_link_libraries(ui PUBLIC services domain)
```

```
add_executable(server src/app/main.cpp)
find_package(Wt REQUIRED COMPONENTS Wt HTTP)
target_link_libraries(server PRIVATE ui services domain Wt::Wt Wt::HTTP)
```

This structure makes it difficult to accidentally couple domain logic to UI or database code.

5.3.11 Operational readiness as a first-class architectural concern

A project-driven architecture treats operational concerns as modules, not afterthoughts:

- configuration is explicit and environment-specific,
- logging is consistent across all layers,
- error handling is structured (domain errors vs UI messages),
- the application lifecycle (startup/shutdown) is deterministic,
- secrets and TLS settings are not hardcoded into code paths.

This is how a Wt project becomes deployable, not just runnable.

5.3.12 Summary

A scalable, project-driven Wt architecture is built around stable boundaries, strict dependency direction, modular targets, and use-case centered growth. The UI is a presentation layer implemented with widgets; controllers orchestrate workflows; services implement business behavior; domain models define meaning; data and infrastructure remain behind interfaces. A composition root near `main()` wires shared services and creates per-session application instances cleanly. This structure prevents architectural collapse, supports testing and team scaling, and aligns naturally with Wt's stateful server-side runtime.

5.4 File and directory organization

5.4.1 Purpose of disciplined organization

In a production-grade Wt application, file and directory organization is an architectural tool, not a cosmetic choice. A clear structure enforces separation of concerns, reduces coupling, improves onboarding, and prevents architectural decay as the project grows in size, scope, and team count.

Because Wt applications are long-lived C++ server processes with stateful sessions, poor organization quickly leads to unmaintainable code where UI logic, business rules, persistence, and infrastructure concerns are mixed. A disciplined layout ensures that growth happens by adding new files and directories, not by rewriting existing ones.

5.4.2 Top-level project structure

A scalable Wt project should start with a stable top-level layout that rarely changes:

```
project-root/
  CMakeLists.txt
  cmake/
  config/
  assets/
  src/
  tests/
  scripts/
  docs/
```

Each directory has a single, well-defined responsibility.

5.4.3 Build system support (`CMakeLists.txt` and `cmake/`)

The top-level `CMakeLists.txt` acts as the orchestration layer for the entire build. It should not contain business logic or file lists beyond adding subdirectories.

```
cmake_minimum_required(VERSION 3.20)
project(wt_project LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_subdirectory(src/domain)
add_subdirectory(src/services)
add_subdirectory(src/data)
add_subdirectory(src/controllers)
add_subdirectory(src/ui)
add_subdirectory(src/app)
```

The `cmake/` directory contains reusable CMake modules, toolchain helpers, and shared configuration logic. This prevents duplication and keeps build logic out of source directories.

5.4.4 Configuration directory (`config/`)

All runtime configuration must be externalized. The `config/` directory typically contains:

```
config/
  wt_config.dev.xml
  wt_config.prod.xml
  app.dev.yaml
  app.prod.yaml
```

This separation ensures:

- environment-specific behavior without recompilation,
- safe deployment practices,
- clear operational boundaries.

Configuration files must never be embedded inside UI or service code.

5.4.5 Static assets (`assets/`)

Static files used by the UI belong in a dedicated directory:

```
assets/
  css/
    main.css
    theme.css
  images/
    logo.png
    icons/
  fonts/
```

These assets are referenced by the UI layer but are not part of the C++ compilation process. In production, they are often served by a reverse proxy.

5.4.6 Source directory (`src/`)

The `src/` directory is strictly organized by architectural layers:

```
src/
  app/
```

```
  ui/
  controllers/
  services/
  domain/
  data/
  infra/
```

Each subdirectory represents a conceptual boundary that should also be enforced at the build level.

5.4.7 Application entry point (`src/app/`)

This directory contains only composition and startup code:

- `main.cpp`
- WServer initialization
- entry point registration
- dependency wiring (composition root)

No business rules, UI widgets, or persistence code belong here.

5.4.8 UI layer (`src/ui/`)

The UI layer contains all widget and page composition code:

```
src/ui/
  pages/
  widgets/
  layout/
  navigation/
```

Responsibilities:

- widget construction and layout,
- event wiring,
- rendering and styling hooks.

The UI layer must not depend on database code or infrastructure utilities.

5.4.9 Controller layer (`src/controllers/`)

Controllers translate UI intent into use-case execution:

- input boundary validation,
- workflow orchestration,
- mapping results to UI-friendly outcomes.

Controllers depend on services and domain models, never on persistence implementations.

5.4.10 Service layer (`src/services/`)

Services implement business use cases and policies:

```
src/services/
  user_service.cpp
  case_service.cpp
  audit_service.cpp
```

Rules:

- no Wt headers,

- no UI assumptions,
- depend only on domain models and repository interfaces.

This ensures services are testable without running the server.

5.4.11 Domain layer (`src/domain/`)

The domain layer defines meaning:

- entities,
- value objects,
- domain commands and results,
- enums and invariants.

The domain layer must be completely independent of Wt, databases, and infrastructure.

5.4.12 Data access layer (`src/data/`)

Persistence logic is isolated here:

```
src/data/
  repositories/
  dbo/
  migrations/
```

This layer implements repository interfaces and depends on database clients or ORMs. It must never leak into UI or controllers.

5.4.13 Infrastructure layer (`src/infra/`)

Infrastructure contains cross-cutting technical concerns:

```
src/infra/
  config/
  logging/
  util/
  time/
```

Examples include configuration loaders, logging adapters, time abstractions, and utility helpers.

5.4.14 Tests directory (`tests/`)

Tests mirror the source structure:

```
tests/
  domain/
  services/
  controllers/
  ui/
  data/
```

This parallel layout reinforces architectural boundaries and improves traceability.

5.4.15 Scripts and automation (`scripts/`)

The `scripts/` directory contains operational tooling:

- build helpers,

- deployment scripts,
- database migration runners,
- environment setup scripts.

5.4.16 Documentation (`docs/`)

The `docs/` directory stores design knowledge:

- architectural decision records,
- onboarding documentation,
- operational notes.

5.4.17 Summary

A clean file and directory organization is foundational to a production-ready Wt application. Top-level directories separate build logic, configuration, assets, source code, tests, scripts, and documentation. The source tree is structured by architectural layers that enforce dependency direction and responsibility boundaries. This organization prevents architectural collapse, supports long-term scaling, and aligns naturally with Wts stateful, server-side execution model.

The Application Class in Depth

6.1 Responsibilities of `WApplication`

6.1.1 What `WApplication` is in the Wt execution model

`Wt::WApplication` is the central session object in Wt. In a typical Wt deployment, the server process hosts many concurrent users; for each active user session, Wt constructs exactly one `WApplication` instance (via the entry point factory). That instance becomes the authoritative owner of:

- the session UI tree (widgets under `root()`),
- session-scoped state and navigation,
- event dispatch for the session,
- incremental rendering and synchronization to the browser.

The architecture consequence is decisive: `WApplication` is not a stateless request handler. It is a long-lived per-session object that coordinates presentation and session lifecycle.

6.1.2 Primary responsibilities

1) Own and manage the root widget tree

The most visible responsibility of `WApplication` is providing the root container for the UI and owning the widget hierarchy:

```
class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("My Wt App");
    root()->addWidget(std::make_unique<Wt::WText>("Hello"));
  }
};
```

Key points:

- `root()` is the top-level container for the session UI.
- Widgets are owned by containers via `std::unique_ptr` transfers.
- When the session ends, the widget tree is destroyed deterministically.

2) Represent session identity and session-scoped state

Because `WApplication` is per session, it is the natural home for *session context* such as:

- authenticated user identity (user id, roles, tenant),
- per-session preferences (language, theme),
- workflow state (which step the user is on),

- per-session caches (small, bounded, derived data).

A professional rule is to store only bounded, session-appropriate state in `WApplication`. Large datasets and long-lived domain objects should remain in services and repositories, loaded on demand.

3) Provide access to the request/session environment

`WApplication` is constructed with a `Wt::WEnvironment` object, which provides session and request context. Practically, this is where you obtain information such as:

- client details (user agent, headers),
- URL parameters and initial request information,
- locale/language preferences,
- deployment-specific settings relevant to rendering.

```
explicit MyApp(const Wt::WEnvironment& env)
  : Wt::WApplication(env)
{
  // Example usage patterns (exact environment APIs depend on your Wt version)
  // const auto agent = env.userAgent();
  // const auto lang  = env.locale().name();
}
```

The environment is primarily used at construction time to initialize the session correctly, but may also inform runtime decisions.

4) Coordinate event dispatch for the session

Wt applications are event-driven. User interactions (clicks, input changes) generate events that are routed back to the server and dispatched into the current session.
`WApplication` is responsible for holding the widget tree and signal connections that determine what code runs when events occur.

```
auto btn = root()->addWidget(std::make_unique<Wt::WPushButton>("Save"));
btn->clicked().connect([this] {
  // Session-safe: this handler runs in the context of this session
  statusText_->setText("Saved");
});
```

Architecturally:

- the UI layer connects signals to handlers,
- handlers should delegate domain work to controllers/services,
- `WApplication` is the session boundary for these dispatches.

5) Maintain and synchronize application state to the browser

A defining responsibility of `WApplication` is coordinating how server-side state changes become browser updates. In Wt, you modify widgets on the server; Wt computes and delivers the necessary updates to the client.
This responsibility includes:

- initial page rendering,
- incremental updates after events,
- keeping UI consistent with server-side widget state.

From a design perspective, this is why Wt behaves more like a server-hosted GUI framework than a template-rendered MVC stack.

6) Manage navigation and internal application paths

A production Wt application needs a navigation model. Wt supports internal paths and session navigation concepts so the user can:

- move between pages,
- deep-link into views,
- use browser navigation semantics where appropriate.

A typical architecture uses `WApplication` as the owner of the navigation system and delegates rendering of pages to a router or navigation component.

```
void showDashboard() {
  root()->clear();
  root()->addWidget(std::make_unique<DashboardPage>(controller_));
}

void showLogin() {
  root()->clear();
  root()->addWidget(std::make_unique<LoginPage>(controller_));
}
```

The navigation mechanism may be implemented with internal paths, a page stack, or a state machine, but `WApplication` is where the session-level decision typically lives.

7) Configure session-level presentation settings

`WApplication` commonly configures presentation properties such as:

- application title,
- theme/style class configuration,
- locale and formatting policy,
- resource loading strategy (CSS, JS resources through Wt mechanisms).

```
setTitle("Production App");
root()->setStyleClass("main-layout");
```

In a professional system, these are driven by configuration and user preferences, not hardcoded constants.

8) Provide a safe place for session-bound timers and periodic tasks

Many applications need periodic UI refresh (dashboards, monitoring, notifications). Wt provides timer mechanisms that are naturally scoped to a session.
A typical pattern:

- session timer triggers,
- handler refreshes a small part of the widget tree,
- heavy work is delegated to services and kept bounded.

```
#include <Wt/WTimer.h>

auto timer = root()->addChild(std::make_unique<Wt::WTimer>());
timer->setInterval(std::chrono::seconds(5));
```

```
timer->timeout().connect([this] {
  refreshCounters();
});
timer->start();
```

6.1.3 Secondary responsibilities that appear in real systems

Session-level dependency access (composition pattern)

A production system typically injects shared services into the session application instance (configuration, logging, DB pool facade, domain services). `WApplication` then becomes the access point for session-safe usage of these dependencies.

```
struct AppServices {
  UserService userService;
  AuditService auditService;
  // plus config/logging facades, repositories, etc.
};

class MyApp final : public Wt::WApplication {
public:
  MyApp(const Wt::WEnvironment& env, AppServices& services)
    : Wt::WApplication(env), services_(services)
  {}

private:
  AppServices& services_;
};
```

This avoids global singletons and keeps dependencies explicit.

Centralized error-to-UI policy

You want one policy for how errors are shown to users. `WApplication` is a natural place to provide:

- a session-scoped notification area,
- a standard error dialog,
- mapping from domain error codes to user messages.

```
void showError(std::string msg) {
  if (!toast_) toast_ = root()->addWidget(std::make_unique<Wt::WText>());
  toast_->setText(std::move(msg));
}
```

6.1.4 What `WApplication` should *not* do

A production-ready architecture requires discipline. `WApplication` should not become the "god object".
Avoid putting these into `WApplication`:

- business rule implementations,
- SQL statements and transaction code,
- repository implementations,
- heavy caches or large datasets per session,
- long blocking operations inside event handlers.

Instead:

- keep `WApplication` responsible for session UI and session orchestration,
- delegate domain work to services and repositories,
- keep per-session state minimal and bounded.

6.1.5 A production template: thin `WApplication`, explicit router, explicit services

The following pattern is a scalable baseline:

```
class AppRouter {
public:
  explicit AppRouter(Wt::WContainerWidget& root) : root_(root) {}

  void showHome() {
    root_.clear();
    root_.addWidget(std::make_unique<Wt::WText>("Home"));
  }

private:
  Wt::WContainerWidget& root_;
};

class MyApp final : public Wt::WApplication {
public:
  MyApp(const Wt::WEnvironment& env, AppServices& services)
    : Wt::WApplication(env), services_(services), router_(*root())
  {
    setTitle("Production App");
```

```
    router_.showHome();
  }

private:
  AppServices& services_;
  AppRouter router_;
};
```

This preserves responsibilities:

- `WApplication`: session lifecycle + session UI ownership
- router: navigation/view composition
- services: business behavior

6.1.6 Summary

`Wt::WApplication` is the per-session core object that owns the root widget tree, holds session context, dispatches events, and coordinates incremental synchronization of UI state to the browser. It also commonly hosts session navigation, presentation configuration, session timers, and a consistent error-to-UI policy. In production systems, `WApplication` must remain thin and must not absorb domain logic or persistence. When kept disciplined, it becomes a stable session boundary that enables scalable architecture, testable business logic, and reliable long-running operation.

6.2 Themes and CSS integration

6.2.1 Why theming matters in a Wt production system

In Wt, user interfaces are constructed from server-side widgets and synchronized to the browser. This can create a false impression that styling is secondary or "automatic". In production, styling is an architectural concern because it affects:

- usability and information clarity,
- consistency across pages and widgets,
- accessibility and readability,
- long-term maintainability (style changes without C++ rewrites),
- deployment (assets, caching, versioning).

A disciplined approach separates **structure and behavior** (C++ widget tree) from **presentation** (themes and CSS). This section explains how to integrate CSS and theming in a way that scales.

6.2.2 The two styling layers in Wt

Styling in Wt is typically managed through two complementary layers:

1. **Theme selection:** a theme defines base widget appearance and widget-to-CSS-class mappings.
2. **CSS integration:** custom stylesheets refine layout, branding, and component-specific presentation.

The theme provides a consistent foundation. Your CSS provides the application identity and fine control.

6.2.3 Theme usage as an application-level responsibility

Theme selection is best configured at the application level because it must be consistent across all pages and widgets in a session. The `WApplication` class is therefore the correct place to:

- choose a theme,
- register global stylesheets,
- set root container classes,
- define global UI layout conventions.

This ensures that every view starts with a consistent styling baseline.

6.2.4 Setting a theme in `WApplication`

A typical pattern is to configure the theme inside the `WApplication` constructor. The exact theme class depends on the theme you choose, but the architectural idea is stable:

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WContainerWidget.h>
#include <memory>

class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Production App");
    root()->setStyleClass("app-root");
```

```
  }
};
```

Theme configuration typically occurs near the start of the constructor to ensure all subsequently created widgets inherit the proper styling conventions.

6.2.5 Adding stylesheets (CSS integration)

A production Wt application normally loads one or more stylesheets that define:

- overall typography and spacing,
- grid/layout primitives,
- color system and brand identity,
- reusable component classes,
- page-specific overrides.

The application should register stylesheets early. A common approach is:

```
#include <Wt/WApplication.h>

class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    useStyleSheet("css/main.css");
    useStyleSheet("css/theme.css");
    root()->setStyleClass("app-root");
  }
};
```

This approach keeps styling out of C++ logic while allowing controlled integration.

6.2.6 Organizing CSS for long-term scalability

A scalable CSS organization avoids “one giant file” and instead uses a layered approach:

```
assets/css/
  base.css
  theme.css
  layout.css
  components/
    buttons.css
    forms.css
    tables.css
  pages/
    login.css
    dashboard.css
```

Recommended responsibilities:

- `base.css`: typography defaults, resets, spacing variables
- `theme.css`: color palette, brand rules, widget overrides
- `layout.css`: grid, containers, responsive primitives
- `components/*.css`: reusable component styles
- `pages/*.css`: page-level customization where necessary

In production, you may combine and minify these assets, but the source organization remains modular.

6.2.7 Mapping widgets to CSS classes

Wt widgets expose style customization through CSS class assignment. The typical approach is:

- assign semantic classes (meaningful names),
- avoid inline styling unless required,
- define consistent naming conventions for components.

Example:

```
auto header = root()->addWidget(std::make_unique<Wt::WText>("Dashboard"));
header->setStyleClass("page-title");

auto card = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
card->setStyleClass("card");
```

This keeps styling declarative. The widget tree declares *what* the element is, and CSS decides *how* it looks.

6.2.8 A minimal CSS baseline for Wt applications

A production system benefits from a small baseline CSS that defines typography and a stable layout container. A minimal example:

```
.app-root {
  font-family: system-ui, -apple-system, "Segoe UI", Arial, sans-serif;
  margin: 0;
  padding: 16px;
}
```

```
.page-title {
  font-size: 1.4rem;
  font-weight: 600;
  margin-bottom: 12px;
}

.card {
  border: 1px solid #e2e2e2;
  border-radius: 8px;
  padding: 16px;
  margin-top: 12px;
}
```

This CSS is intentionally simple and compatible with progressive enhancement.

6.2.9 Theme-friendly component design

To keep components theme-friendly:

- prefer semantic classes (e.g., `btn-primary`, `form-field`, `card`),
- avoid hardcoding colors into C++ (do it in CSS),
- use consistent spacing and typography tokens in CSS,
- ensure components are reusable (widgets map to CSS, not to unique one-off styles).

This allows theme changes without changing C++ UI composition logic.

6.2.10 Serving CSS assets in development and production

There are two common production-friendly approaches:

Approach A: Wt serves assets

- simplest during development,
- consistent behavior across environments,
- easier portability (single binary + assets).

Approach B: Reverse proxy serves assets

- preferred for high-performance production systems,
- better caching and compression control,
- isolates static delivery from application runtime.

A project-driven architecture should support both: Wt-serving for local development and proxy-serving for production.

6.2.11 Caching and versioning strategy

Production deployments must avoid stale CSS due to browser caching. A robust strategy includes:

- versioned asset filenames (e.g., `main.v123.css`),
- or query-string versioning (e.g., `main.css?v=123`),
- long cache headers for static assets (handled by reverse proxy),
- predictable rebuild pipeline that updates versions on release.

This prevents UI inconsistencies after deployments.

6.2.12 Accessibility and readability as first-class styling requirements

In production-grade systems, CSS must support:

- adequate contrast and readable typography,
- focus styles for keyboard navigation,
- responsive layout that handles different screen sizes,
- predictable spacing that improves comprehension.

Even in enterprise and internal systems, accessibility failures can become operational risks and support costs.

6.2.13 A professional pattern: a UI theme module

Large projects treat theme/CSS integration as a module. A common design is:

- a `UiTheme` class that loads stylesheets and sets root classes,
- a single call in `WApplication` constructor to apply it.

```
class UiTheme {
public:
  static void apply(Wt::WApplication& app) {
    app.useStyleSheet("css/base.css");
    app.useStyleSheet("css/layout.css");
    app.useStyleSheet("css/theme.css");
    app.root()->setStyleClass("app-root");
  }
};
```

```
class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    UiTheme::apply(*this);
    setTitle("Production App");
  }
};
```

This makes theming reproducible, testable as a configuration unit, and easy to maintain.

6.2.14 Summary

Themes and CSS integration in Wt must be treated as a core architectural concern. The `WApplication` class is the correct place to establish theme and global CSS policy, while widgets should only declare semantic CSS classes. A scalable system uses modular CSS organization, avoids inline styling, supports caching/versioning strategies for production deployments, and enforces accessibility and consistency. By separating UI behavior (C++ widgets and events) from presentation (theme and CSS), you keep the codebase maintainable and enable evolution of the applications visual identity without rewriting core logic.

6.3 Session management

6.3.1 Why session management is a core architectural topic in Wt

Wt is fundamentally a **stateful, session-based** web platform. Unlike classic stateless request/response frameworks (where each request reconstructs state), Wt keeps a **server-side**

UI object graph per user session. This yields powerful capabilities (desktop-like interaction, incremental updates), but it also means:

- session lifetime directly impacts memory usage and scalability,
- session correctness affects security (authentication, authorization, replay),
- multi-instance deployments require deliberate routing and affinity decisions,
- background work must be synchronized safely with session-bound UI state.

For production systems, session management is not an implementation detail. It is an operational contract.

6.3.2 The Wt session model in one sentence

In the common Wt execution model:

> **One browser session corresponds to one server-side `Wt::WApplication` instance, which owns a persistent widget tree and session-scoped state.**

This is the mental model that drives every architectural decision in a Wt project.

6.3.3 What constitutes "session state" in a production Wt application

A production system must define what is allowed to live in session memory. A disciplined definition typically includes:

- identity and authorization context (user id, roles, tenant),
- UI navigation state (current page/view, view parameters),
- transient workflow state (wizard step, selected item id),

- small bounded caches (derived values, last query filter),
- CSRF-like per-session tokens (if you implement them at the app layer).

And it explicitly excludes:

- large datasets (lists of thousands of rows),
- long-lived domain entities that can go stale,
- database connections owned per session,
- global caches stored inside session objects.

The rule is simple: **store identifiers and small derived state, not heavy data.**

6.3.4 Session lifetime: creation, activity, expiration, teardown

Creation

A session is typically created when the user first accesses an application entry point and the server calls your factory function to construct a `WApplication` instance.

```
server.addEntryPoint(Wt::EntryPointType::Application,
  [](const Wt::WEnvironment& env) {
    return std::make_unique<MyApp>(env);
  });
```

Activity

During the session:

- user actions generate events,
- events are dispatched into the session's application instance,
- UI updates are synchronized back to the browser incrementally.

Expiration

Sessions should expire after inactivity to free memory and reduce security risk. In production, the expiration/timeout policy is a capacity and security decision:

- shorter timeouts reduce memory and exposure window,
- longer timeouts improve user convenience but increase resource footprint.

Teardown

When a session ends (explicit logout, expiration, server shutdown), the `WApplication` instance and its widget tree are destroyed deterministically. Any resources referenced by widgets or session state must be safe to release at this time.

6.3.5 Where to implement session policy: configuration vs code

Professional systems separate **policy** from **mechanics**:

- policy (timeouts, cookie rules, deployment settings) belongs in configuration,
- mechanics (session context object, authorization checks, logout flow) belongs in code.

This avoids recompilation for operational adjustments and allows environment-specific tuning (dev vs prod).

6.3.6 A session context object: the minimal professional baseline

Treat session identity and authorization as a first-class object, not scattered fields. The following is a minimal, production-friendly example:

```cpp
#include <string>
#include <optional>
#include <vector>

struct SessionContext {
  std::optional<std::string> userId;
  std::vector<std::string> roles;
  std::optional<std::string> tenantId;

  bool isAuthenticated() const noexcept { return userId.has_value(); }

  bool hasRole(const std::string& r) const {
    for (const auto& x : roles) if (x == r) return true;
    return false;
  }
};
```

You then store exactly one instance of `SessionContext` in `WApplication` and pass a const reference to controllers/services when needed.

6.3.7 Authentication lifecycle: login, session binding, logout

Login

Login should produce a session-bound identity context:

- validate credentials (service),
- establish `SessionContext` (application),
- navigate to the authorized area (router/navigation).

```
class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {}

  void onLoginSuccess(std::string userId,
                      std::vector<std::string> roles,
                      std::string tenantId)
  {
    session_.userId = std::move(userId);
    session_.roles = std::move(roles);
    session_.tenantId = std::move(tenantId);
    showHome();
  }

  void logout()
  {
    session_ = SessionContext{};
    root()->clear();
    showLogin();
  }

private:
  void showHome();
  void showLogin();

  SessionContext session_;
};
```

Logout

Logout must:

- clear session-bound identity,
- clear UI state (root content and navigation),
- optionally invalidate any app-level tokens you issue,
- return the user to a safe public view.

A common mistake is to only change the UI without clearing session identity data.

6.3.8 Authorization: never trust UI-only protection

In Wt, it is easy to hide widgets when the user lacks a role. That is necessary for UX, but insufficient for security. Authorization must also be enforced in services:

- UI: hide/disable actions the user cannot perform,
- Controller/Service: enforce the rule before mutating state or data.

A service-level check pattern:

```
struct AuthContext {
  std::string userId;
  std::vector<std::string> roles;

  bool hasRole(const std::string& r) const {
    for (const auto& x : roles) if (x == r) return true;
    return false;
  }
```

```
};

class AdminService {
public:
  bool canAccessAdmin(const AuthContext& a) const {
    return a.hasRole("admin");
  }
};
```

The UI should never be the final enforcement point.

6.3.9 Concurrency and session safety

One session, many events

Session code is event-driven. While a single session is logically isolated, a production server handles many sessions concurrently. This implies:

- per-session state is safe when only accessed in that session's event context,
- shared resources (caches, pools, loggers) must be thread-safe,
- any background worker that touches session UI must synchronize correctly.

Background work: do not update widgets from arbitrary threads

A common production need is doing long work (reports, external calls). The safe architecture is:

1. perform heavy work in a worker (thread pool / async),
2. store results in a thread-safe structure,

3. schedule a session-safe update that modifies the widget tree.

Even when the framework provides mechanisms to post work back to the session, the architectural rule remains:

> **Only update session widgets from the session's safe execution context.**

6.3.10 Scaling to multiple server instances

Many Wt deployments eventually move from one server process to multiple instances behind a reverse proxy or load balancer. The session-based model imposes constraints:

Sticky sessions (session affinity)

Because the server holds session state, the simplest multi-instance deployment uses session affinity:

- the load balancer routes a given user session consistently to the same backend instance,
- sessions remain in the memory of that instance.

This is operationally straightforward and common for stateful server applications.

Why naive round-robin breaks

If requests from the same session are routed to different backends without shared session state:

- the backend receiving the request may not have the session in memory,
- the user experiences resets, lost UI state, or forced re-logins,
- correctness and security assumptions fail.

Therefore, you must choose either:

- session affinity, or
- a deliberate strategy for shared session state (more complex), or
- an architecture that avoids server-held session UI state (not Wt's default strength).

6.3.11 Memory discipline: the production survival rule

The session model makes memory discipline mandatory. A production team should define:

- maximum acceptable session memory footprint,
- timeouts and idle expiration policies,
- limits for data cached per session,
- safe patterns for pagination and incremental loading.

A reliable approach for lists and tables is:

- store only filter parameters and selected ids in session state,
- fetch paged data through services on demand,
- never store entire large result sets in the session widget tree.

6.3.12 A minimal "session-aware" application skeleton

The following skeleton shows a disciplined placement of session management responsibilities:

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
```

```
#include <memory>
#include <string>
#include <vector>
#include <optional>

struct SessionContext {
  std::optional<std::string> userId;
  std::vector<std::string> roles;

  bool isAuthenticated() const noexcept { return userId.has_value(); }
};

class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    root()->setStyleClass("app-root");
    if (session_.isAuthenticated()) showHome();
    else showLogin();
  }

  void loginAs(std::string userId, std::vector<std::string> roles)
  {
    session_.userId = std::move(userId);
    session_.roles = std::move(roles);
    showHome();
  }
```

```
  void logout()
  {
    session_ = SessionContext{};
    showLogin();
  }

  const SessionContext& session() const noexcept { return session_; }

private:
  void showLogin()
  {
    root()->clear();
    root()->addWidget(std::make_unique<Wt::WText>("Login view"));
  }

  void showHome()
  {
    root()->clear();
    root()->addWidget(std::make_unique<Wt::WText>("Home view"));
  }

  SessionContext session_;
};
```

This skeleton is intentionally minimal, but it demonstrates the correct boundaries:

- `WApplication` owns session context and navigation decisions,
- UI views are created under `root()`,
- authentication changes session context explicitly,

- logout clears both identity and UI.

6.3.13 Summary

Session management in Wt is central because Wt is stateful and session-driven: a `WApplication` instance and widget tree exist per user session, and session lifetime directly affects memory, scalability, and security. A production architecture defines exactly what belongs in session state (identity, workflow, small caches), enforces service-level authorization, uses explicit logout and expiration policies, and respects concurrency rules by updating widgets only from the session-safe context. Multi-instance deployments must account for server-held session state, commonly using session affinity. With disciplined session management, Wt becomes a stable foundation for production-grade systems rather than a fragile stateful prototype.

6.4 Working with environment and request data

6.4.1 What “environment and request data” means in Wt

In a traditional web framework, request data is extracted from the HTTP request and passed to handlers for each route. In Wt, because the runtime hosts long-lived sessions with persistent `WApplication` instances, environment and request data are provided through the `Wt::WEnvironment` object when the session is created and through per-event context during interactions. Understanding how to correctly access and use this data is essential for:

- reading query parameters and form data,
- inspecting headers, cookies, and client metadata,
- making routing or rendering decisions based on the request,

- obtaining security-relevant tokens (CSRF, session identifiers),
- supporting localization, content negotiation, and mobile/desktop variants.

6.4.2 The role of `Wt::WEnvironment`

The `Wt::WEnvironment` object encapsulates environment and request-specific context at the moment the session is created. It provides strongly typed access to:

- request headers (e.g., user agent, accept-language),
- query parameters,
- cookies sent with the initial request,
- remote address and protocol information,
- session identifiers and security tokens,
- server configuration values relevant to the request.

Because sessions can persist over many events, `WEnvironment` is most useful at the point of session construction or when handling explicit navigation events.

6.4.3 Accessing environment data during construction

A typical pattern is to inspect the environment inside the `WApplication` constructor:

```
class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
```

```
    // Example: extract the client's preferred language
    auto clientLocale = env.locale().name();
    // Example: read a query parameter
    auto paramValue = env.getParameter("ref");
    // Example: inspect a header
    auto userAgent = env.getHeader("User-Agent");
  }
};
```

6.4.4 Query parameters and navigation context

Query parameters are part of the URL used to initiate the session and can be read via `getParameter()`:

```
auto page     = env.getParameter("page");
auto filter   = env.getParameter("filter");
auto sortMode = env.getParameter("sort");
```

These values can influence the initial state of your UI (for example, which tab to show first or which filters to apply).

6.4.5 Cookies: reading and writing

Wt provides access to cookies from the environment, and also allows setting cookies from the application. Cookies are important for:

- session identifiers when not using a separate session store,
- feature flags,
- user preference storage,

- lightweight tracking where appropriate.

```
auto cookieOpt = env.getCookie("theme");
if (cookieOpt) {
  auto theme = *cookieOpt;
  // use theme preference
}

// Setting a cookie
setCookie("theme", "dark", /*expires*/ 365 * 24 * 60 * 60);
```

Always treat cookies as untrusted input and validate or encode values accordingly.

6.4.6 Headers and client metadata

The environment provides header access. This is useful for:

- logging and analytics,
- mobile/desktop content decisions,
- feature detection,
- simple security heuristics (e.g., enforcing a required referer).

```
auto acceptLang = env.getHeader("Accept-Language");
auto userAgent  = env.getHeader("User-Agent");
```

Be mindful that headers can be spoofed and should not be used as a sole security boundary.

6.4.7 Remote address and transport information

In some applications, you need to know the clients IP or protocol (HTTP vs HTTPS). Wt exposes this through the environment:

```
auto remoteAddress = env.remoteAddress();
bool isSecure      = env.scheme() == "https";
```

In production behind reverse proxies, you must manage `X-Forwarded-For` and related headers carefully (often at the reverse proxy or load balancer level) and configure Wt accordingly so that `remoteAddress()` reflects the true client address.

6.4.8 Form data and widget-bound inputs

Much of the form and input data in Wt applications is bound at the widget layer. For example, a `WLineEdit` holds its current text value in memory. When handling events, you read widget values directly rather than parsing raw HTTP form payloads.

```
auto nameValue = nameEdit_->text().toUTF8();
auto emailValue = emailEdit_->text().toUTF8();
```

Unlike stateless frameworks that reparse form data per request, Wts widgets retain state and make it available in event handlers.

6.4.9 URL fragments and internal navigation

Wt supports internal URL fragments or path changes to represent session navigation. This can be integrated with the environment at session creation and later with navigation APIs.
When your application reads a fragment like `#section`, you can use:

```
auto fragment = env.getParameter("fragment"); // or similar helper
```

Then initialize your page accordingly. For deeper navigation integration, you may combine environment data with your own routing layer.

6.4.10 Handling POST/PUT/DELETE semantics

While much of Wts interaction model is event-driven and does not expose raw HTTP verbs directly, you can inspect the HTTP method from the environment:

```
auto method = env.requestMethod();
if (method == "POST") {
  // handle a form in a specialized way
}
```

In practice, Wts widget-based event model abstracts many of these details. Explicit handling of HTTP methods is only needed for cases like file uploads, REST endpoints, or integration with external systems using non-idempotent actions.

6.4.11 Security considerations when using environment data

- always treat environment-derived values (parameters, headers, cookies) as untrusted,
- validate and sanitize before using in logic or database queries,
- never use client-supplied URLs directly for internal navigation without validation,
- log potentially suspicious patterns for monitoring.

For example, if extracting a numeric identifier from a query parameter:

```
auto idParam = env.getParameter("id");
if (!idParam.empty()) {
  try {
    auto id = std::stoll(idParam);
    // use id safely
  } catch (...) {
```

```
    // handle invalid input
  }
}
```

6.4.12 Localization and request context

The environment indicates client language preferences. This can drive localized UI or messages:

```
auto localeName = env.locale().name();
// e.g., "en_US" or "fr_FR"
```

Professional systems use this information to select resource bundles or formatting policies, keeping localization logic separate from widget code.

6.4.13 A session-aware request abstraction

For larger systems, it is often helpful to wrap environment access into a small abstraction that decouples your application logic from Wt-specific APIs:

```
struct RequestContext {
  std::string remoteIp;
  std::string userAgent;
  std::string locale;
  std::map<std::string,std::string> queryParams;
};
```

You populate this once and pass it to controllers/services, avoiding accidental scattering of environment reads throughout the UI code.

6.4.14 Summary

Working with environment and request data in Wt is a core part of building session-aware, secure, and responsive applications. The `Wt::WEnvironment` object provided at session creation contains query parameters, headers, remote address, cookies, and locale information. In a professional architecture, you read and validate this data early, convert it to a safe internal form, and pass it to controllers and services. Direct interaction with widget values happens in event handlers, not through low-level parsing of HTTP payloads. Always validate untrusted input, manage proxy-related headers correctly, and separate environment access from business logic for maintainability and security.

Part IV

Building the User Interface (UI Layer)

Core Wt Widgets

7.1 Containers

7.1.1 What a container means in Wt

In Wt, a *container* is a widget whose primary job is to **own, arrange, and manage** other widgets. Containers form the backbone of the UI tree:

- every interactive UI is ultimately a hierarchy rooted at `WApplication::root()`,
- containers are the nodes that group children and control structure,
- layout policies (vertical, horizontal, grid, border) are usually attached to containers,
- ownership is deterministic: when a container is destroyed, its children are destroyed.

A professional Wt UI is not "a list of widgets". It is a carefully composed container tree with stable structural boundaries and predictable update behavior.

7.1.2 Core ownership rule: containers own their children

Wt UI composition follows an explicit ownership model. When you add a widget using a `std::unique_ptr`, the container becomes the owner:

```
auto c = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
c->addWidget(std::make_unique<Wt::WText>("Child"));
```

Professional consequences:

- do not `delete` UI objects manually,
- avoid storing raw pointers unless lifetime is clearly owned by a container,
- keep containers small and structured; large unstructured trees become hard to maintain.

7.1.3 The fundamental container: `WContainerWidget`

`Wt::WContainerWidget` is the general-purpose container used almost everywhere. It represents a structural grouping element in the rendered page and is the default choice when you need to:

- group widgets into a section,
- attach CSS classes to a block,
- apply a layout manager,
- clear/replace a portion of the UI.

Creating containers and using semantic CSS classes

A production UI should create containers with semantic meaning:

```
auto page = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
page->setStyleClass("page");

auto header = page->addWidget(std::make_unique<Wt::WContainerWidget>());
```

```
header->setStyleClass("page-header");

auto body = page->addWidget(std::make_unique<Wt::WContainerWidget>());
body->setStyleClass("page-body");
```

This pattern scales because CSS controls appearance while C++ controls structure and behavior.

Clearing and rebuilding a sub-tree

A container can be treated as a view region that is replaced:

```
void showMessage(Wt::WContainerWidget& region, const std::string& msg) {
  region.clear();
  region.addWidget(std::make_unique<Wt::WText>(msg));
}
```

This is the foundation for simple routers, wizard steps, and page transitions.

7.1.4 Layout-based containers: structuring space and alignment

Most professional UIs do not rely on implicit flow. They attach layout managers to containers. Common layout patterns include:

- vertical stacking (forms, dashboards),
- horizontal grouping (toolbars, button rows),
- grid placement (tables of inputs, admin panels),
- border layouts (header/body/footer),
- stacked layouts (page switching).

Vertical layout (`WVBoxLayout`)

```
#include <Wt/WVBoxLayout.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

auto form = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
form->setStyleClass("form");

auto v = form->setLayout(std::make_unique<Wt::WVBoxLayout>());
v->addWidget(std::make_unique<Wt::WText>("Username"));
auto user = v->addWidget(std::make_unique<Wt::WLineEdit>());
v->addWidget(std::make_unique<Wt::WText>("Password"));
auto pass = v->addWidget(std::make_unique<Wt::WLineEdit>());
pass->setEchoMode(Wt::EchoMode::Password);
v->addWidget(std::make_unique<Wt::WPushButton>("Sign in"));
```

Horizontal layout (`WHBoxLayout`)

```
#include <Wt/WHBoxLayout.h>

auto toolbar = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
toolbar->setStyleClass("toolbar");

auto h = toolbar->setLayout(std::make_unique<Wt::WHBoxLayout>());
h->addWidget(std::make_unique<Wt::WPushButton>("New"));
h->addWidget(std::make_unique<Wt::WPushButton>("Edit"));
h->addWidget(std::make_unique<Wt::WPushButton>("Delete"));
```

Grid layout (`WGridLayout`)

```
#include <Wt/WGridLayout.h>

auto gridBox = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
gridBox->setStyleClass("grid-form");

auto g = gridBox->setLayout(std::make_unique<Wt::WGridLayout>());
g->addWidget(std::make_unique<Wt::WText>("First name"), 0, 0);
g->addWidget(std::make_unique<Wt::WLineEdit>(),          0, 1);
g->addWidget(std::make_unique<Wt::WText>("Last name"),   1, 0);
g->addWidget(std::make_unique<Wt::WLineEdit>(),          1, 1);
```

7.1.5 Structured containers: common UI building blocks

Beyond generic containers, Wt provides higher-level containers that embed a specific UI structure. In production, these reduce boilerplate and encourage consistent patterns.

Group containers (`WGroupBox`)

A group box is a titled region useful for forms and settings pages:

```
#include <Wt/WGroupBox.h>

auto account = root()->addWidget(std::make_unique<Wt::WGroupBox>("Account"));
account->addWidget(std::make_unique<Wt::WText>("Email"));
account->addWidget(std::make_unique<Wt::WLineEdit>());
```

Panels and collapsible regions (`WPanel`)

Panels are useful for dashboards and administrative screens where sections may expand/collapse:

```
#include <Wt/WPanel.h>

auto p = root()->addWidget(std::make_unique<Wt::WPanel>());
p->setTitle("Advanced settings");

auto content = std::make_unique<Wt::WContainerWidget>();
content->addWidget(std::make_unique<Wt::WText>("Option A"));
content->addWidget(std::make_unique<Wt::WLineEdit>());
p->setCentralWidget(std::move(content));
```

Stacked containers for view switching (`WStackedWidget`)

When implementing navigation without full page reloads, stacked widgets are the simplest reliable primitive:

```
#include <Wt/WStackedWidget.h>

auto stack = root()->addWidget(std::make_unique<Wt::WStackedWidget>());

auto page1 = stack->addWidget(std::make_unique<Wt::WContainerWidget>());
page1->addWidget(std::make_unique<Wt::WText>("Home"));

auto page2 = stack->addWidget(std::make_unique<Wt::WContainerWidget>());
page2->addWidget(std::make_unique<Wt::WText>("Settings"));

stack->setCurrentWidget(page1); // switch to page2 when needed
```

Production tip: treat each page as its own widget/component and let a router decide which page is active.

Tabbed containers (`WTabWidget`)

Tabs are appropriate for dense admin screens or settings panels:

```
#include <Wt/WTabWidget.h>

auto tabs = root()->addWidget(std::make_unique<Wt::WTabWidget>());

auto general = std::make_unique<Wt::WContainerWidget>();
general->addWidget(std::make_unique<Wt::WText>("General settings"));
tabs->addTab(std::move(general), "General");

auto security = std::make_unique<Wt::WContainerWidget>();
security->addWidget(std::make_unique<Wt::WText>("Security settings"));
tabs->addTab(std::move(security), "Security");
```

7.1.6 Template containers: separating structure from content

For teams that prefer HTML-like structure while keeping logic in C++, template-based containers can be used to:

- define a stable UI skeleton,
- bind widgets into named placeholders,
- reduce layout boilerplate in C++.

A typical pattern is using a template widget and binding placeholders:

```
#include <Wt/WTemplate.h>

auto t = root()->addWidget(std::make_unique<Wt::WTemplate>());
```

```
t->setTemplateText(
  "<div class='page'>"
  "  <h1>${title}</h1>"
  "  <div class='content'>${content}</div>"
  "</div>",
  Wt::TextFormat::XHTML
);

t->bindWidget("title", std::make_unique<Wt::WText>("Dashboard"));
t->bindWidget("content", std::make_unique<Wt::WText>("Welcome"));
```

Use templates when structure is stable and the team wants a clear separation of layout skeleton and dynamic widget content.

7.1.7 Performance and scalability considerations

Containers are not free. Each widget participates in the server-side tree and in synchronization. A production UI should:

- keep the widget tree reasonably sized,
- avoid rebuilding large subtrees on minor updates,
- use view switching (stacked widgets) instead of destroying and recreating heavy views,
- paginate and virtualize data-heavy screens (do not render thousands of rows as widgets).

7.1.8 Testing and maintainability guidelines

To keep container usage professional:

- define page widgets as classes, not giant constructors in `WApplication`,

- ensure each page has a clear root container and a narrow public API,
- name CSS classes semantically and keep styling out of C++,
- treat containers as architectural boundaries (page regions, components).

7.1.9 Summary

Containers are the structural foundation of Wt user interfaces. `WContainerWidget` provides general composition, layouts provide deterministic arrangement, and higher-level containers such as group boxes, panels, stacked widgets, and tab widgets enable professional UI patterns with less boilerplate. A scalable Wt UI is a well-designed container tree: semantic structure in C++, presentation in CSS, and controlled updates that keep session memory and synchronization costs bounded.

7.2 Input widgets

7.2.1 Why input widgets are a production-critical topic

Input widgets are the boundary where untrusted user data enters your system. In production-grade Wt applications, input widgets are not “UI controls” only; they are the front line for:

- correctness (type-safe parsing, consistent validation),
- security (rejecting malformed input, preventing injection patterns),
- usability (clear feedback, predictable focus behavior),
- maintainability (reusable field components and validation policies).

Because Wt is stateful and event-driven, input widgets also shape your architecture: they define when validation happens, how errors are displayed, and how workflows progress.

7.2.2 Core principles for input handling in Wt

A professional Wt project should apply these principles consistently:

- **Always validate at two levels:** UI validation for feedback, and service validation for enforcement.
- **Never trust widget values:** treat all user input as untrusted, even if it comes from a widget state.
- **Prefer typed widgets and validators:** avoid ad-hoc parsing scattered across handlers.
- **Separate formatting from meaning:** UI formatting is not domain validation.
- **Centralize error display policy:** consistent UX reduces support cost.

7.2.3 Text input: `WLineEdit`

`Wt::WLineEdit` is the standard single-line text input. It is used for usernames, short labels, emails, ids, and search fields.

Basic usage

```
#include <Wt/WLineEdit.h>
#include <Wt/WText.h>
#include <Wt/WPushButton.h>
#include <Wt/WContainerWidget.h>

auto box = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
```

```
auto name = box->addWidget(std::make_unique<Wt::WLineEdit>());
name->setPlaceholderText("User name");

auto status = box->addWidget(std::make_unique<Wt::WText>());

auto submit = box->addWidget(std::make_unique<Wt::WPushButton>("Submit"));
submit->clicked().connect([=] {
  const std::string value = name->text().toUTF8();
  status->setText("Value length: " + std::to_string(value.size()));
});
```

Password input

For passwords, configure echo mode:

```
name->setEchoMode(Wt::EchoMode::Password);
```

In production, treat password handling as sensitive:

- do not log raw passwords,
- avoid storing them in long-lived strings longer than necessary,
- send them only to a dedicated authentication service.

Signal patterns for text inputs

Common signals:

- `changed()` for committed change,
- `keyWentUp()` or `keyPressed()` for reactive search patterns,

- `enterPressed()` for forms.

A typical “press Enter to submit” pattern:

```
name->enterPressed().connect([=] {
  submit->click();
});
```

7.2.4 Multi-line input: `WTextArea`

`Wt::WTextArea` is used for longer input such as descriptions, messages, and notes.

```
#include <Wt/WTextArea.h>

auto desc = root()->addWidget(std::make_unique<Wt::WTextArea>());
desc->setRows(6);
desc->setPlaceholderText("Description");
```

Production guidance:

- enforce max length at the UI boundary,
- validate for domain constraints in services,
- use consistent encoding conversions when persisting.

7.2.5 Numeric and typed input: `WSpinBox`, `WDoubleSpinBox`

For numeric fields, prefer typed widgets to reduce parsing errors.

Integer input

```
#include <Wt/WSpinBox.h>

auto age = root()->addWidget(std::make_unique<Wt::WSpinBox>());
age->setRange(0, 150);
age->setValue(30);
```

Floating point input

```
#include <Wt/WDoubleSpinBox.h>

auto price = root()->addWidget(std::make_unique<Wt::WDoubleSpinBox>());
price->setRange(0.0, 1000000.0);
price->setDecimals(2);
price->setValue(99.99);
```

Typed widgets help eliminate ad-hoc `std::stod` parsing scattered in UI handlers.

7.2.6 Boolean input: `WCheckBox`

`Wt::WCheckBox` represents a boolean state.

```
#include <Wt/WCheckBox.h>

auto agree =
↪  root()->addWidget(std::make_unique<Wt::WCheckBox>("I accept the terms"));
```

In production, treat checkbox values as untrusted and enforce critical policies in services.

7.2.7 Single choice: `WRadioButton` and `WButtonGroup`

Radio buttons are used for mutually exclusive choices.

```
#include <Wt/WRadioButton.h>
#include <Wt/WButtonGroup.h>

auto group = std::make_unique<Wt::WButtonGroup>();

auto r1 = root()->addWidget(std::make_unique<Wt::WRadioButton>("Option A"));
auto r2 = root()->addWidget(std::make_unique<Wt::WRadioButton>("Option B"));

group->addButton(r1, 1);
group->addButton(r2, 2);
r1->setChecked(true);
```

7.2.8 Drop-down selection: `WComboBox`

`Wt::WComboBox` provides a compact selection list.

```
#include <Wt/WComboBox.h>

auto country = root()->addWidget(std::make_unique<Wt::WComboBox>());
country->addItem("Saudi Arabia");
country->addItem("UAE");
country->addItem("Qatar");

country->changed().connect([=] {
  auto selected = country->currentText().toUTF8();
});
```

Production guidance:

- map UI items to stable internal identifiers (not display strings),
- do not trust client-provided indices without validation,
- localize display strings without changing internal ids.

7.2.9 Date and time input

Wt provides widgets for structured date/time input. Production guidance:

- store timestamps in a consistent canonical representation in the backend,
- avoid mixing local time and UTC without explicit policy,
- validate date ranges and business rules in services.

A typical usage pattern:

```
#include <Wt/WDateEdit.h>

auto dob = root()->addWidget(std::make_unique<Wt::WDateEdit>());
dob->setFormat("yyyy-MM-dd");
```

7.2.10 File upload: `WFileUpload`

File upload is a high-risk input channel and must be handled carefully.

```
#include <Wt/WFileUpload.h>
#include <Wt/WPushButton.h>

auto upload = root()->addWidget(std::make_unique<Wt::WFileUpload>());
```

```
upload->setFileTextSize(60);

auto btn = root()->addWidget(std::make_unique<Wt::WPushButton>("Upload"));
btn->clicked().connect([=] {
  upload->upload();
});
```

Production requirements for uploads:

- enforce size limits at the server and reverse proxy,
- validate MIME type and file signatures (not only extension),
- store uploaded files outside the web root,
- generate safe internal names (no user-supplied paths),
- scan or quarantine if required by policy.

7.2.11 Validation: widgets + validators

Wt supports validators that attach to input widgets. A professional approach uses validators for immediate feedback while still enforcing the same rules in services.
A typical pattern:

```
#include <Wt/WRegExpValidator.h>

auto email = root()->addWidget(std::make_unique<Wt::WLineEdit>());
email->setPlaceholderText("email@example.com");

auto v = std::make_shared<Wt::WRegExpValidator>(".+@.+\\..+");
v->setMandatory(true);
email->setValidator(v);
```

Production guidance:

- keep UI validators simple and user-friendly,
- never rely on regex-only validation for security,
- use domain validation in services as the final authority.

7.2.12 Form handling: consistent boundary validation

A production architecture avoids scattered validation in multiple click handlers by using a form component or helper that:

- reads all field values,
- runs widget validators,
- produces a command object,
- calls a controller/service method,
- displays errors consistently.

Example of boundary validation flow:

```
bool validateAndShow(Wt::WLineEdit& field, Wt::WText& errorOut) {
  if (field.validate() == Wt::ValidationState::Valid) {
    errorOut.setText("");
    return true;
  }
  errorOut.setText(field.validationResult().message());
  return false;
}
```

7.2.13 Security discipline for input widgets

Input widgets reduce low-level parsing work, but they do not eliminate security risks. Production discipline includes:

- validate, normalize, and encode before persistence,
- parameterize database queries (never concatenate),
- apply output encoding rules for any user-supplied content that is rendered,
- enforce authorization before performing actions triggered by input,
- rate-limit sensitive actions (login, password reset) at a higher layer.

7.2.14 Maintainability: building reusable field components

As projects scale, you should avoid copying the same input wiring patterns. A professional approach is to define reusable components such as:

- `LabeledField` (label + widget + error text),
- `ValidatedLineEdit` (line edit + validator + error output),
- `FormSection` containers with consistent CSS.

This reduces duplication, enforces consistent UX, and centralizes validation behavior.

7.2.15 Summary

Input widgets in Wt provide typed, stateful components for collecting user data: `WLineEdit` and `WTextArea` for text, spin boxes for numbers, checkboxes and radio groups for choices, combo boxes for selections, date editors for structured dates, and file upload widgets for binary

input. In production systems, input handling must be disciplined: use validators for immediate feedback, enforce domain rules in services, treat all input as untrusted, centralize error display policy, and build reusable field components to keep large UI codebases maintainable and consistent.

7.3 Buttons

7.3.1 Why buttons deserve serious engineering attention

Buttons look simple, but in production systems they are **command boundaries**. A button click usually means:

- a state transition (save, submit, approve, delete),
- a security-sensitive action (login, privilege change),
- a potentially expensive operation (report generation, export),
- a workflow milestone (step completion).

In Wt, a button click is delivered as an event into the server-side session where you can modify state and UI immediately. That power requires discipline: button handlers must remain thin, delegate to controllers/services, and provide consistent feedback.

7.3.2 Core button widgets in Wt

Wt provides several button-related widgets, each intended for specific cases:

- `Wt::WPushButton` for the standard clickable action button.
- `Wt::WToolBar` + buttons for command bars (when used).

- `Wt::WRadioButton` / `Wt::WCheckBox` (selection inputs, not commands).
- `Wt::WAnchor` when you need link semantics (navigation).

For command actions, `WPushButton` is the main production workhorse.

7.3.3 `WPushButton`: the standard command button

Creating a button and handling clicks

```
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
#include <Wt/WContainerWidget.h>

auto box = root()->addWidget(std::make_unique<Wt::WContainerWidget>());

auto status = box->addWidget(std::make_unique<Wt::WText>());

auto save = box->addWidget(std::make_unique<Wt::WPushButton>("Save"));
save->clicked().connect([=] {
  status->setText("Save requested");
});
```

Production rule: the click handler is not the business logic

In production, the handler should delegate to a controller/service:

```
save->clicked().connect([=] {
  auto r = controller_.onSaveDraft();
  status->setText(r.ok ? "Saved" : "Save failed");
});
```

This keeps UI concerns (widgets, rendering) separate from domain decisions and persistence.

7.3.4 Disabling buttons and preventing double-submit

Double-submit is a common production failure mode, especially for operations that take time (DB writes, network calls). A professional pattern is:

- disable the button immediately,
- show progress feedback,
- re-enable only on completion (success or failure).

```
auto submit = root()->addWidget(std::make_unique<Wt::WPushButton>("Submit"));
auto status = root()->addWidget(std::make_unique<Wt::WText>());

submit->clicked().connect([=] {
  submit->setDisabled(true);
  status->setText("Working...");

  auto r = controller_.onSubmit(); // keep it fast; no blocking UI thread if heavy
  if (r.ok) status->setText("Done");
  else {
    status->setText("Failed");
    submit->setDisabled(false);
  }
});
```

If the work is heavy, move it to a background worker and post the result back to the session safely.

7.3.5 Default buttons and keyboard interaction

Forms often need a default action when the user presses Enter. A practical pattern is wiring input widgets to trigger the button click:

```
nameEdit_->enterPressed().connect([=] { submit->click(); });
passwordEdit_->enterPressed().connect([=] { submit->click(); });
```

This improves usability and makes your Wt app feel like a professional UI rather than a sequence of mouse-only actions.

7.3.6 Buttons as navigation vs buttons as commands

Do not use a command button for navigation when link semantics are more appropriate. In Wt, navigation can be implemented via:

- internal path changes (router navigation),
- `WAnchor` for link-like behavior,
- a button that calls your router only when the UX is truly command-like.

A clean navigation button pattern:

```
auto back = root()->addWidget(std::make_unique<Wt::WPushButton>("Back"));
back->clicked().connect([this] { router_.showPrevious(); });
```

But if the action is purely navigation, a link-style element may be more semantically correct depending on the UI design.

7.3.7 Styling buttons with CSS classes

In production, styling must be declarative. Do not build custom colors and spacing in code. Assign semantic CSS classes:

```
auto primary = root()->addWidget(std::make_unique<Wt::WPushButton>("Create"));
primary->setStyleClass("btn btn-primary");

auto danger = root()->addWidget(std::make_unique<Wt::WPushButton>("Delete"));
danger->setStyleClass("btn btn-danger");
```

A minimal CSS convention:

```
.btn { padding: 8px 12px; border-radius: 8px; }
.btn-primary { font-weight: 600; }
.btn-danger { font-weight: 600; }
```

The important point is semantic naming and consistent reuse, not the specific visual design.

7.3.8 Icon buttons and compact controls

Many enterprise UIs use icon-only buttons in toolbars. The correct pattern is:

- keep accessibility in mind (tooltips/labels),
- use CSS classes to size and align,
- do not hide critical meaning behind an unlabeled icon.

```
auto refresh = root()->addWidget(std::make_unique<Wt::WPushButton>("Refresh"));
refresh->setStyleClass("btn btn-icon");
refresh->setToolTip("Refresh data");
```

7.3.9 Confirmation for destructive actions

A production system must treat destructive actions as high-risk operations. A typical approach:

- show a confirmation dialog,

- perform authorization check in services,
- log or audit if policy requires.

A practical pattern using a message box:

```
#include <Wt/WMessageBox.h>

auto del = root()->addWidget(std::make_unique<Wt::WPushButton>("Delete"));
del->setStyleClass("btn btn-danger");

del->clicked().connect([=] {
  auto box = std::make_unique<Wt::WMessageBox>(
    "Confirm",
    "Delete this item?",
    Wt::Icon::Warning,
    Wt::StandardButton::Yes | Wt::StandardButton::No
  );

  auto* raw = box.get();
  raw->buttonClicked().connect([=] (Wt::StandardButton b) {
    if (b == Wt::StandardButton::Yes) {
      controller_.onDelete(); // must enforce auth in service layer too
    }
    raw->removeFromParent();
  });

  root()->addChild(std::move(box));
});
```

The dialog is a UX protection layer. The service layer remains the enforcement layer.

7.3.10 Progress feedback and long operations

Buttons often trigger actions that involve:

- database writes,
- external API calls,
- report generation,
- file processing.

The production rule is:

> Do not block the session thread for long tasks; show progress and complete asynchronously.

At minimum, provide immediate UI feedback and disable the initiating button. For long tasks, move work to a background worker and safely schedule UI updates back into the session context.

7.3.11 Button-driven form validation workflow

A standard professional workflow:

- user clicks the button,
- UI runs widget validation and displays field-level errors,
- controller builds a command object and calls the service,
- service enforces domain rules and authorization,
- UI displays success/failure summary consistently.

Example of a compact boundary validation helper:

```
bool validateAll(std::initializer_list<Wt::WFormWidget*> fields) {
  bool ok = true;
  for (auto* f : fields) {
    if (f->validate() != Wt::ValidationState::Valid) ok = false;
  }
  return ok;
}
```

7.3.12 Maintainability: reusable button patterns

As the UI grows, define reusable patterns:

- `PrimaryButton` builder that assigns classes and disables on click,
- `DangerButton` builder with confirmation policy,
- `AsyncActionButton` wrapper that handles busy state.

This prevents repeating fragile logic across dozens of pages.

7.3.13 Summary

Buttons in Wt represent command boundaries: a click triggers server-side session logic and often causes state changes. `WPushButton` is the primary widget for commands; it must be wired to thin handlers that delegate to controllers and services. Production-grade handling includes disabling to prevent double-submit, consistent validation and error display, semantic CSS classes for styling, confirmations for destructive actions, and progress feedback for long-running operations. With disciplined patterns, buttons remain safe, predictable, and scalable as the application grows.

7.4 Layouts

7.4.1 Why layouts are the real foundation of professional Wt UIs

In Wt, containers define the widget tree, but **layouts** define how that tree becomes a usable interface. A production UI must remain readable and consistent across:

- different browsers,
- different screen sizes and DPI settings,
- varying content length (localization, user-provided text),
- growing feature sets and expanding forms,
- long-lived maintenance where UI evolves without breaking alignment.

Layouts are therefore not a convenience API. They are a stability mechanism. A disciplined layout strategy prevents fragile UIs built from ad-hoc spacing and uncontrolled CSS hacks.

7.4.2 Two layout approaches in Wt

Wt supports two complementary ways of organizing UI:

1. **Widget layout managers** (C++ side): `WVBoxLayout`, `WHBoxLayout`, `WGridLayout`, `WBorderLayout`, `WStackedWidget`-based patterns.
2. **CSS-driven layout** (browser side): flexbox/grid/responsive CSS applied via `setStyleClass()` and stylesheets.

Production systems usually use *both*:

- layout managers for deterministic widget composition and alignment,
- CSS for branding, spacing tokens, responsive behavior, and visual polish.

7.4.3 The layout ownership rule

Layouts belong to containers. A container typically receives exactly one layout manager:

```
#include <Wt/WContainerWidget.h>
#include <Wt/WVBoxLayout.h>

auto page = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
auto v = page->setLayout(std::make_unique<Wt::WVBoxLayout>());
```

Once a layout is set, child widgets should be added through the layout, not directly to the container, to ensure alignment rules are applied consistently.

7.4.4 Vertical layout: `WVBoxLayout`

`WVBoxLayout` stacks widgets vertically. It is the default choice for:

- forms,
- settings pages,
- wizard steps,
- dashboards with stacked panels.

Example: a clean form layout

```
#include <Wt/WText.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WVBoxLayout.h>
```

```
auto form = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
form->setStyleClass("form");

auto v = form->setLayout(std::make_unique<Wt::WVBoxLayout>());
v->addWidget(std::make_unique<Wt::WText>("Email"));
auto email = v->addWidget(std::make_unique<Wt::WLineEdit>());
email->setPlaceholderText("email@example.com");

v->addWidget(std::make_unique<Wt::WText>("Password"));
auto pass = v->addWidget(std::make_unique<Wt::WLineEdit>());
pass->setEchoMode(Wt::EchoMode::Password);

auto submit = v->addWidget(std::make_unique<Wt::WPushButton>("Sign in"));
submit->setStyleClass("btn btn-primary");
```

Spacing and stretch

A production UI often needs consistent spacing between rows. Prefer CSS classes for spacing tokens, but you can also control layout spacing where supported.
If your layout supports spacing configuration, set it once at the layout level rather than inserting dummy widgets.

7.4.5 Horizontal layout: `WHBoxLayout`

`WHBoxLayout` places widgets in a single horizontal row. It is commonly used for:

- toolbars,
- button rows,
- compact filters (search + dropdown + action).

Example: toolbar

```
#include <Wt/WHBoxLayout.h>

auto bar = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
bar->setStyleClass("toolbar");

auto h = bar->setLayout(std::make_unique<Wt::WHBoxLayout>());
h->addWidget(std::make_unique<Wt::WPushButton>("New"))->setStyleClass("btn");
h->addWidget(std::make_unique<Wt::WPushButton>("Edit"))->setStyleClass("btn");
h->addWidget(std::make_unique<Wt::WPushButton>("Delete"))->setStyleClass("btn btn-danger")
```

Production advice:

- do not pack too many controls into one row,
- use groups and separators for readability,
- keep destructive actions visually distinct.

7.4.6 Grid layout: `WGridLayout`

`WGridLayout` is essential for structured forms and complex panels. It is the most reliable way to align labels and fields.

Example: label-field grid

```
#include <Wt/WGridLayout.h>
#include <Wt/WText.h>
#include <Wt/WLineEdit.h>

auto box = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
```

```
box->setStyleClass("grid-form");

auto g = box->setLayout(std::make_unique<Wt::WGridLayout>());

g->addWidget(std::make_unique<Wt::WText>("First name"), 0, 0);
auto first = g->addWidget(std::make_unique<Wt::WLineEdit>(), 0, 1);

g->addWidget(std::make_unique<Wt::WText>("Last name"), 1, 0);
auto last = g->addWidget(std::make_unique<Wt::WLineEdit>(), 1, 1);

g->addWidget(std::make_unique<Wt::WText>("Email"), 2, 0);
auto email = g->addWidget(std::make_unique<Wt::WLineEdit>(), 2, 1);
```

Production advice:

- keep grids small and local (per form section),
- avoid mega-grids spanning entire pages,
- keep column meaning consistent (labels left, fields right).

7.4.7 Border layout: `WBorderLayout`

Border layouts help structure a classic page skeleton:

- header (north),
- sidebar (west),
- main content (center),
- optional footer (south).

Example: application shell

```
#include <Wt/WBorderLayout.h>

auto shell = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
shell->setStyleClass("app-shell");

auto bl = shell->setLayout(std::make_unique<Wt::WBorderLayout>());

auto header = std::make_unique<Wt::WContainerWidget>();
header->setStyleClass("app-header");
bl->addWidget(std::move(header), Wt::LayoutPosition::North);

auto sidebar = std::make_unique<Wt::WContainerWidget>();
sidebar->setStyleClass("app-sidebar");
bl->addWidget(std::move(sidebar), Wt::LayoutPosition::West);

auto content = std::make_unique<Wt::WContainerWidget>();
content->setStyleClass("app-content");
bl->addWidget(std::move(content), Wt::LayoutPosition::Center);
```

This establishes a stable structure. Page content changes inside the center container, not by rebuilding the whole shell.

7.4.8 Stacked layouts: view switching without rebuilding

Wt supports page switching patterns using stacked containers. The goal is to avoid destroying and recreating heavy views unnecessarily.

Example: `WStackedWidget` navigation

```
#include <Wt/WStackedWidget.h>
#include <Wt/WText.h>

auto stack = root()->addWidget(std::make_unique<Wt::WStackedWidget>());
stack->setStyleClass("page-stack");

auto home = stack->addWidget(std::make_unique<Wt::WContainerWidget>());
home->addWidget(std::make_unique<Wt::WText>("Home"));

auto settings = stack->addWidget(std::make_unique<Wt::WContainerWidget>());
settings->addWidget(std::make_unique<Wt::WText>("Settings"));

stack->setCurrentWidget(home);
// later: stack->setCurrentWidget(settings);
```

Production advice:

- use stacked widgets for high-level pages or major view regions,
- keep each page as its own class to prevent huge constructors,
- clear only what must change; keep shell layout stable.

7.4.9 Layout + CSS: the professional hybrid approach

Layout managers determine structure, but CSS controls:

- spacing system (margins/padding tokens),
- typography,

- responsive behavior,
- alignment tweaks (center, stretch, wrap),
- consistent component appearance.

A production pattern is to apply semantic classes to containers and let CSS do final presentation:

```
shell->setStyleClass("app-shell");
```

Minimal CSS example:

```
.app-shell { padding: 16px; }
.app-header { margin-bottom: 12px; }
.app-sidebar { width: 240px; }
.app-content { padding: 12px; }
```

This keeps the C++ layout stable while allowing the visual system to evolve without code changes.

7.4.10 Responsive design strategy

Even enterprise/internal systems are increasingly used on laptops, tablets, and phones. A practical strategy:

- build a stable layout skeleton (border layout or container composition),
- rely on CSS media queries for responsive changes,
- avoid hardcoding pixel-perfect dimensions in C++,
- prefer scalable units and CSS-driven breakpoints.

If responsiveness is a requirement, do not attempt to solve it only with C++ layout managers. Use CSS for the last mile.

7.4.11 Performance and maintainability guidelines

- prefer a small number of predictable layout regions,
- avoid deep widget nesting when a simpler structure works,
- do not rebuild large UI trees for small changes,
- keep layout decisions local (page-level and component-level),
- build reusable layout components (shell, card, form section).

7.4.12 Summary

Layouts in Wt define how containers and widgets become a usable interface. `WVBoxLayout` supports vertical composition, `WHBoxLayout` supports toolbars and rows, `WGridLayout` provides aligned forms and structured panels, and `WBorderLayout` creates stable application shells with header/sidebar/content regions. For scalable production UIs, combine layout managers with semantic CSS classes and stylesheets: keep structure in C++, presentation in CSS, and use stacked widgets for view switching without excessive rebuilding. This hybrid approach yields maintainable, consistent, and responsive interfaces suitable for production-grade Wt systems.

7.5 Events and signals

7.5.1 Why the event model is the core of Wt

Wt is not built around controller actions per HTTP request. It is built around an event-driven model that resembles desktop UI frameworks:

- the browser is the remote rendering surface,

- the server holds the authoritative widget tree per session,
- user interactions produce events,
- events trigger C++ handlers connected to widget signals,
- handlers update server-side state and widgets,
- Wt synchronizes the minimal necessary changes back to the browser.

Understanding events and signals is therefore essential for designing maintainable UIs, preventing concurrency mistakes, and building production-grade workflows.

7.5.2 Signals in Wt: the conceptual model

A **signal** represents "something happened" (clicked, changed, selected). A **slot** is a function/callback connected to that signal.
In Wt, signals are strongly typed and are typically connected using:

- lambdas,
- member function pointers,
- `std::bind` patterns (less preferred in modern C++).

7.5.3 The basic pattern: connect a handler

Lambda connection

```
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
#include <Wt/WContainerWidget.h>
```

```
auto status = root()->addWidget(std::make_unique<Wt::WText>());

auto btn = root()->addWidget(std::make_unique<Wt::WPushButton>("Click"));
btn->clicked().connect([=] {
  status->setText("Clicked");
});
```

Member function connection

When you want clean separation and reuse, connect signals to class member functions:

```
class Page final : public Wt::WContainerWidget {
public:
  Page() {
    status_ = addWidget(std::make_unique<Wt::WText>());
    btn_ = addWidget(std::make_unique<Wt::WPushButton>("Save"));
    btn_->clicked().connect(this, &Page::onSaveClicked);
  }

private:
  void onSaveClicked() {
    status_->setText("Saving...");
  }

  Wt::WText* status_{nullptr};
  Wt::WPushButton* btn_{nullptr};
};
```

This is typically the best style for production pages and reusable widgets.

7.5.4 Signal categories you use constantly

Button and action signals

- `WPushButton::clicked()`
- menu item triggers (when using menus or toolbars)

Text input signals

For `WLineEdit` and similar widgets:

- `changed()` when an edit is committed,
- `textInput()` for live input behavior (use carefully),
- `enterPressed()` for form submit patterns.

Example:

```
auto edit = root()->addWidget(std::make_unique<Wt::WLineEdit>());
edit->setPlaceholderText("Search");

edit->changed().connect([=] {
  auto q = edit->text().toUTF8();
  // trigger a search using q
});
```

Selection signals

For selection widgets such as `WComboBox`:

- `changed()` to respond to selection changes

```
auto cb = root()->addWidget(std::make_unique<Wt::WComboBox>());
cb->addItem("Low");
cb->addItem("High");

cb->changed().connect([=] {
  auto selection = cb->currentIndex();
});
```

Timer signals

Timers enable periodic UI refresh within a session:

```
#include <Wt/WTimer.h>

auto timer = root()->addChild(std::make_unique<Wt::WTimer>());
timer->setInterval(std::chrono::seconds(2));
timer->timeout().connect([=] {
  // refresh a section
});
timer->start();
```

7.5.5 Event dispatch and session context

Every signal handler executes in the context of a specific session. This has major consequences:

- you can safely update widgets for that session from inside handlers,
- per-session state stored in `WApplication` or page objects is directly available,
- you must not assume handlers are global or shared across sessions.

In other words, signal handlers are per-session logic, not global request handlers.

7.5.6 Avoiding the most common production mistake: lifetime capture bugs

Signals often capture variables by reference in lambdas. In a stateful UI, the lifetime of objects matters. A safe professional rule:

- do not capture references to objects that may be destroyed,
- capture `this` only when the widget owning the signal outlives the connection,
- store persistent child widget pointers as members if you must access them later.

Example of a safe pattern (member-owned widgets):

```
class LoginPage final : public Wt::WContainerWidget {
public:
  LoginPage() {
    user_ = addWidget(std::make_unique<Wt::WLineEdit>());
    pass_ = addWidget(std::make_unique<Wt::WLineEdit>());
    pass_->setEchoMode(Wt::EchoMode::Password);

    auto btn = addWidget(std::make_unique<Wt::WPushButton>("Login"));
    btn->clicked().connect(this, &LoginPage::onLogin);
  }

private:
  void onLogin() {
    const std::string u = user_->text().toUTF8();
    const std::string p = pass_->text().toUTF8();
    // delegate to controller/service
  }
```

```
  Wt::WLineEdit* user_{nullptr};
  Wt::WLineEdit* pass_{nullptr};
};
```

7.5.7 Signal design: keep handlers thin

A production Wt application should enforce:

- UI handlers perform validation and UI feedback,
- handlers call controllers/services for business behavior,
- services enforce rules and authorization,
- handlers update widgets based on returned results.

A clean pattern:

```
btn->clicked().connect([=] {
  if (edit->validate() != Wt::ValidationState::Valid) {
    status->setText("Invalid input");
    return;
  }

  auto cmd = edit->text().toUTF8();
  auto r = controller_.run(cmd);

  status->setText(r.ok ? "Success" : "Failure");
});
```

7.5.8 Chaining events and composing workflows

Because Wt applications are stateful, workflows are commonly implemented by chaining event handlers that move the user through a state machine:

- Step 1 handler validates input, saves draft state, navigates to Step 2.
- Step 2 handler performs next action, etc.

A production pattern is to:

- keep workflow state in a session context or controller,
- render the current step as a page widget,
- ensure each step is idempotent and safe against repeated clicks.

7.5.9 Asynchronous work and signals: safe UI updates

A critical production rule:

> Do not update widgets from background threads.

If a button click triggers heavy work:

- run heavy work in a worker thread or job system,
- store results in a safe shared structure,
- schedule a UI update back into the session context.

Even if you choose different mechanisms (thread pools, async tasks), the rule remains: UI mutation must occur in the safe session execution context.

7.5.10 Event storms and performance

Some signals can generate high-frequency events (e.g., live typing). In production:

- do not perform heavy queries on every keystroke,
- debounce user input (timer-based delay) for search fields,
- avoid rebuilding large widget subtrees repeatedly,
- update only the minimal region needed.

Debounce pattern:

```
#include <Wt/WTimer.h>

auto search = root()->addWidget(std::make_unique<Wt::WLineEdit>());
auto t = root()->addChild(std::make_unique<Wt::WTimer>());
t->setInterval(std::chrono::milliseconds(300));
t->timeout().connect([=] {
  t->stop();
  auto q = search->text().toUTF8();
  // run search now
});

search->textInput().connect([=] {
  t->stop();
  t->start();
});
```

7.5.11 Event and signal patterns you should standardize

A professional team standardizes:

- a consistent validation policy (when to validate, where to display errors),
- a consistent busy-state policy (disable buttons, show progress),
- consistent error handling (domain error mapping to UI messages),
- consistent navigation triggers (router calls, internal path updates).

This reduces bugs and makes UI behavior predictable.

7.5.12 Summary

Events and signals are the operational heart of Wt. Widgets publish typed signals that represent user interactions and timers, and you connect them to C++ handlers (slots). Handlers run in the session context, enabling direct updates to the server-side widget tree, which Wt then synchronizes incrementally to the browser. Production-grade usage requires discipline: avoid lifetime capture bugs, keep handlers thin, delegate business rules to services, handle long tasks asynchronously with safe session updates, and control event frequency with debouncing and minimal UI updates. When treated as an architectural system rather than “callbacks”, Wts event model enables scalable, maintainable, and highly interactive server-side web applications.

Building a Professional Application Layout

8.1 Header, sidebar, and content areas

8.1.1 Why structured page areas matter in professional UIs

In modern web applications, users expect consistent navigation, clear content focus, and predictable layout behavior. A common professional page structure separates the interface into:

- a persistent **header** (global navigation, branding, user actions),
- an optional **sidebar** (feature navigation, contextual tools),
- a central **content area** (primary task surface),
- optional **footer** (supplemental links, status).

This separation enhances usability, improves cognitive mapping for users, and supports responsive design. In Wt, achieving this layout requires combining layout managers with container hierarchies and clear styling.

8.1.2 The structural pattern: application shell

A scalable layout defines a stable shell that holds header and sidebar once, and replaces only the content area as the user navigates. This prevents unnecessary re-composition of static

regions and keeps application state predictable.

The typical master layout uses a border layout that places regions explicitly:

```
#include <Wt/WContainerWidget.h>
#include <Wt/WBorderLayout.h>
#include <Wt/WText.h>

auto shell = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
shell->setStyleClass("app-shell");

auto layout = shell->setLayout(std::make_unique<Wt::WBorderLayout>());
```

8.1.3 Header area

The header provides global navigation and status information (user identity, application title, global actions).

Adding the header

```
auto header = std::make_unique<Wt::WContainerWidget>();
header->setStyleClass("app-header");

// branding
header->addWidget(std::make_unique<Wt::WText>("My Application"));

// user status
header->addWidget(std::make_unique<Wt::WText>("  |  Logged in as "));
header->addWidget(std::make_unique<Wt::WText>("User Name"));

layout->addWidget(std::move(header), Wt::LayoutPosition::North);
```

Header design principles

- Keep height fixed or minimal growth (avoid scrollable headers).
- Group global actions (help, logout) consistently at one edge.
- Use CSS classes for spacing and alignment, not inline style.

8.1.4 Sidebar area

The sidebar provides contextual navigation: feature groups, modules, or deep links.

Adding the sidebar

```
auto sidebar = std::make_unique<Wt::WContainerWidget>();
sidebar->setStyleClass("app-sidebar");

// example navigation links
sidebar->addWidget(std::make_unique<Wt::WText>("Dashboard"));
sidebar->addWidget(std::make_unique<Wt::WText>("Reports"));
sidebar->addWidget(std::make_unique<Wt::WText>("Settings"));

layout->addWidget(std::move(sidebar), Wt::LayoutPosition::West);
```

Sidebar design principles

- Use semantic CSS classes to control width and overflow.
- Keep the navigation consistent across pages.
- Support collapse/expand patterns for responsive design.
- Avoid embedding large functional code; use links or button events to switch content.

8.1.5 Content area

The content area is where primary UI activities occur: lists, forms, dashboards, etc. It should be flexible and replaceable without disturbing the shell.

Adding the content area

```
auto content = std::make_unique<Wt::WContainerWidget>();
content->setStyleClass("app-content");

layout->addWidget(std::move(content), Wt::LayoutPosition::Center);
```

Content replacement pattern

A professional UI does not destroy the shell to show a new page. Instead, it updates the content area:

```
void showPage(Wt::WContainerWidget& contentRegion, std::unique_ptr<Wt::WWidget>
↪  page) {
  contentRegion.clear();
  contentRegion.addWidget(std::move(page));
}
```

This pattern isolates navigation logic from layout composition.

8.1.6 Putting it all together: a complete shell example

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WBorderLayout.h>
#include <Wt/WPushButton.h>
```

```
#include <Wt/WText.h>

class MainApp : public Wt::WApplication {
public:
  MainApp(const Wt::WEnvironment& env) : Wt::WApplication(env) {
    auto shell = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
    shell->setStyleClass("app-shell");

    auto layout = shell->setLayout(std::make_unique<Wt::WBorderLayout>());

    // header
    auto header = std::make_unique<Wt::WContainerWidget>();
    header->setStyleClass("app-header");
    header->addWidget(std::make_unique<Wt::WText>("My Application"));
    layout->addWidget(std::move(header), Wt::LayoutPosition::North);

    // sidebar
    auto sidebar = std::make_unique<Wt::WContainerWidget>();
    sidebar->setStyleClass("app-sidebar");
    auto dbBtn =
    ↪  sidebar->addWidget(std::make_unique<Wt::WPushButton>("Dashboard"));
    auto settingsBtn =
    ↪  sidebar->addWidget(std::make_unique<Wt::WPushButton>("Settings"));
    layout->addWidget(std::move(sidebar), Wt::LayoutPosition::West);

    // content region
    contentRegion_ = layout->addWidget(std::make_unique<Wt::WContainerWidget>(),
    ↪  Wt::LayoutPosition::Center);
```

```
    // event wiring
    dbBtn->clicked().connect(this, &MainApp::showDashboard);
    settingsBtn->clicked().connect(this, &MainApp::showSettings);

    showDashboard();
  }

private:
  void showDashboard() {
    contentRegion_->clear();
    contentRegion_->addWidget(std::make_unique<Wt::WText>("Dashboard content"));
  }

  void showSettings() {
    contentRegion_->clear();
    contentRegion_->addWidget(std::make_unique<Wt::WText>("Settings content"));
  }

  Wt::WContainerWidget* contentRegion_{nullptr};
};
```

8.1.7 CSS integration for layout regions

Use semantic classes to control spacing, branding, and responsive behavior:

```
.app-shell {
  display: flex;
  flex-direction: column;
  height: 100%;
}
```

```
.app-header {
  background: #f8f9fa;
  padding: 12px 16px;
  border-bottom: 1px solid #ddd;
}

.app-sidebar {
  background: #ffffff;
  border-right: 1px solid #ddd;
  padding: 8px;
  width: 200px;
}

.app-content {
  padding: 16px;
  overflow: auto;
}
```

Professional guidelines:

- prefer semantic CSS classes over widget-level style calls,
- enable responsive adjustments with media queries,
- avoid inline styles that mix presentation with logic.

8.1.8 Responsive and collapsible sidebar patterns

For larger professional systems, sidebars often collapse on narrow screens:

```
@media (max-width: 768px) {
  .app-sidebar {
    display: none;
  }
}
```

A button in the header can toggle the sidebar:

```
auto toggle = header->addWidget(std::make_unique<Wt::WPushButton>(""));
toggle->clicked().connect([=] {
  contentRegion_->toggleStyleClass("sidebar-hidden");
});
```

Responsive behavior is handled by CSS and class toggles rather than C++ layout manipulation.

8.1.9 Accessibility and keyboard navigation

In professional applications:

- ensure the header and sidebar navigation are keyboard reachable,
- use semantic labels for navigation buttons,
- provide ARIA roles if required by compliance guidelines.

For example:

```
dbBtn->setToolTip("Go to dashboard (Alt+D)");
```

8.1.10 Summary

Header, sidebar, and content areas form the stable shell of a professional Wt application layout. Use a border layout manager to define these regions, and treat the content region as replaceable

while keeping the header and sidebar persistent. Apply semantic CSS classes for spacing, alignment, and responsive behavior, and wire navigation events to replace only the content area. This pattern promotes maintainability, a consistent user experience, and clear separation between structure and content in production-grade systems.

8.2 Reusable UI components

8.2.1 Why reuse is mandatory in production UI code

A Wt application grows by adding pages, workflows, tables, dialogs, and forms. If you build each page as a one-off widget tree inside `WApplication`, the codebase will degrade quickly:

- duplicated form wiring and validation logic,
- inconsistent layout and styling across pages,
- fragile event handlers copied with small variations,
- rising cost for every UI change (because the same pattern exists in many files),
- higher defect rate (each copy diverges and breaks differently).

Reusable UI components solve this by making **structure + behavior** a first-class unit. In Wt, a reusable component is typically a custom widget class that:

- encapsulates a small UI subtree,
- exposes a narrow public API,
- owns its internal widgets and signal wiring,
- applies consistent CSS classes and layout rules,
- delegates business logic to controllers/services.

8.2.2 The three layers of reuse in Wt UI architecture

A professional Wt UI usually defines reusable components at three levels:

1) Atomic components

Small building blocks:

- labeled field,
- validation message,
- primary button,
- badge, status pill,
- toolbar row.

2) Composite components

Reusable sections:

- login form,
- search panel with filters,
- confirmation dialog wrapper,
- paged table view.

3) Page components

Complete pages or views:

- dashboard page,

- settings page,
- entity editor page.

The rule: build reuse from small to large, and avoid coupling page components to application entry code.

8.2.3 Component design rules (production discipline)

A reusable UI component should follow these rules:

- **Single responsibility:** one clear purpose (form field, toolbar, dialog).
- **Stable public API:** expose only what callers need (get/set value, set error, on submit).
- **Internal ownership:** child widgets are private and owned by the component.
- **No domain logic inside:** UI collects input and displays results; services enforce rules.
- **Semantic styling:** CSS classes describe role, not visual details.
- **Testability:** logic parts are in controllers/services; UI remains thin.

8.2.4 Pattern: atomic reusable component (`LabeledField`)

A repeated pattern in every professional application is: label + input + error message. Encapsulate it once.

```
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <Wt/WLineEdit.h>
#include <Wt/WVBoxLayout.h>
```

```
class LabeledLineEdit final : public Wt::WContainerWidget {
public:
  explicit LabeledLineEdit(const std::string& labelText,
                           const std::string& placeholder = {})
  {
    setStyleClass("field");

    auto v = setLayout(std::make_unique<Wt::WVBoxLayout>());

    label_ = v->addWidget(std::make_unique<Wt::WText>(labelText));
    label_->setStyleClass("field-label");

    edit_ = v->addWidget(std::make_unique<Wt::WLineEdit>());
    edit_->setPlaceholderText(placeholder);
    edit_->setStyleClass("field-input");

    error_ = v->addWidget(std::make_unique<Wt::WText>());
    error_->setStyleClass("field-error");
    error_->hide();
  }

  Wt::WLineEdit& input() noexcept { return *edit_; }

  std::string valueUtf8() const { return edit_->text().toUTF8(); }

  void setError(const std::string& msg) {
    error_->setText(msg);
    error_->show();
    edit_->addStyleClass("is-invalid");
```

```
  }

  void clearError() {
    error_->setText("");
    error_->hide();
    edit_->removeStyleClass("is-invalid");
  }

private:
  Wt::WText* label_{nullptr};
  Wt::WLineEdit* edit_{nullptr};
  Wt::WText* error_{nullptr};
};
```

Usage:

```
auto emailField = root()->addWidget(std::make_unique<LabeledLineEdit>("Email",
↪  "email@example.com"));
```

This provides consistent structure, consistent CSS hooks, and a stable API for validation feedback.

8.2.5 Pattern: reusable button builders (primary, secondary, danger)

Buttons must be consistent across the application. Create small helpers:

```
#include <Wt/WPushButton.h>

struct UiButtons {
  static std::unique_ptr<Wt::WPushButton> primary(const std::string& text) {
    auto b = std::make_unique<Wt::WPushButton>(text);
```

```
    b->setStyleClass("btn btn-primary");
    return b;
  }

  static std::unique_ptr<Wt::WPushButton> danger(const std::string& text) {
    auto b = std::make_unique<Wt::WPushButton>(text);
    b->setStyleClass("btn btn-danger");
    return b;
  }
};
```

Usage:

```
auto save = root()->addWidget(UiButtons::primary("Save"));
auto del  = root()->addWidget(UiButtons::danger("Delete"));
```

8.2.6 Pattern: composite reusable component (`FormSection`)

A form section often repeats: title + description + fields container. Encapsulate it:

```
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>

class FormSection final : public Wt::WContainerWidget {
public:
  FormSection(const std::string& title, const std::string& help = {}) {
    setStyleClass("form-section");

    auto h = addWidget(std::make_unique<Wt::WText>(title));
    h->setStyleClass("form-section-title");
```

```
    if (!help.empty()) {
      auto p = addWidget(std::make_unique<Wt::WText>(help));
      p->setStyleClass("form-section-help");
    }

    body_ = addWidget(std::make_unique<Wt::WContainerWidget>());
    body_->setStyleClass("form-section-body");
  }

  Wt::WContainerWidget& body() noexcept { return *body_; }

private:
  Wt::WContainerWidget* body_{nullptr};
};
```

Usage:

```
auto account = root()->addWidget(std::make_unique<FormSection>(
  "Account",
  "Manage identity and access settings."
));

account->body().addWidget(std::make_unique<LabeledLineEdit>("Email"));
account->body().addWidget(std::make_unique<LabeledLineEdit>("Phone"));
```

8.2.7 Pattern: page components with a narrow API

A page should be a widget class with:

- constructor takes required dependencies (controllers/services),

- exposes minimal events (signals) to the router,
- internally manages its widget tree and event wiring.

```
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <Wt/WPushButton.h>

class DashboardPage final : public Wt::WContainerWidget {
public:
  explicit DashboardPage() {
    setStyleClass("page dashboard");


    ↪  addWidget(std::make_unique<Wt::WText>("Dashboard"))->setStyleClass("page-title");

    auto refresh = addWidget(std::make_unique<Wt::WPushButton>("Refresh"));
    refresh->setStyleClass("btn btn-secondary");
    refresh->clicked().connect(this, &DashboardPage::onRefresh);
  }

private:
  void onRefresh() {
    // delegate to controller/service in real code
  }
};
```

A router then swaps pages into the content area without rebuilding the entire shell.

8.2.8 Event wiring inside components: avoid leaking internals

A reusable component should connect its signals internally and expose only what the parent needs. Two safe patterns:

Expose high-level signals

For example, a login form exposes `submitted(username,password)` rather than exposing the raw edits.

```
#include <Wt/WSignal.h>

class LoginForm final : public Wt::WContainerWidget {
public:
  Wt::Signal<std::string,std::string>& submitted() { return submitted_; }

  LoginForm() {
    user_ = addWidget(std::make_unique<Wt::WLineEdit>());
    pass_ = addWidget(std::make_unique<Wt::WLineEdit>());
    pass_->setEchoMode(Wt::EchoMode::Password);

    auto btn = addWidget(std::make_unique<Wt::WPushButton>("Login"));
    btn->clicked().connect([this] {
      submitted_.emit(user_->text().toUTF8(), pass_->text().toUTF8());
    });
  }

private:
  Wt::Signal<std::string,std::string> submitted_;
  Wt::WLineEdit* user_{nullptr};
```

```
  Wt::WLineEdit* pass_{nullptr};
};
```

Expose setters/getters for state

For example, a search panel exposes `query()` and `setBusy(bool)`.

8.2.9 Styling reusable components: semantic CSS contract

Reusable components must have a stable CSS contract:

- the component sets CSS classes that describe roles (field, error, toolbar),
- CSS defines spacing, typography, colors, responsiveness,
- parents should not depend on internal widget structure.

Example minimal CSS:

```
.field { margin-bottom: 12px; }
.field-label { font-weight: 600; margin-bottom: 6px; display: block; }
.field-input { width: 100%; }
.field-error { margin-top: 6px; font-size: 0.9rem; }
.is-invalid { border: 1px solid #d00; }
```

8.2.10 Reusable components and validation policy

A consistent validation policy is one of the highest-impact reuse areas. Recommended approach:

- field components encapsulate UI feedback (error text, invalid CSS),
- controllers/services own the validation rules,

- UI maps service validation errors to field errors using stable field keys.

A simple mapping pattern:

```
struct ValidationError {
  std::string field;   // e.g., "email"
  std::string message; // human readable
};
```

Then a form component can route errors to the correct field widget.

8.2.11 Reusable components and performance

In Wt, components live as server-side widget subtrees. Production guidance:

- do not create extremely deep trees when a simpler structure works,
- reuse page shells and replace only content,
- avoid rebuilding entire components on small state changes,
- keep component creation cheap and deterministic.

8.2.12 Recommended component directory organization

A clean architecture keeps reusable UI components in a dedicated module:

```
src/ui/
  components/
    LabeledLineEdit.h
    FormSection.h
    ConfirmDialog.h
```

```
    Buttons.h
  pages/
    DashboardPage.h
    SettingsPage.h
  layout/
    AppShell.h
    Router.h
```

This reinforces reuse and keeps pages from duplicating component logic.

8.2.13 Summary

Reusable UI components are essential for production-grade Wt applications. The professional approach is to build custom widget classes that encapsulate structure, event wiring, and consistent CSS hooks while exposing only a narrow public API. Reuse should exist at multiple levels: atomic components (fields, buttons), composite components (form sections, dialogs), and page components (dashboard, settings). Keep business rules out of UI components, delegate to controllers/services, and standardize validation and feedback behavior across the application. With disciplined reuse, the UI layer remains scalable, consistent, and maintainable as the project grows.

8.3 Dynamic UI updates

8.3.1 What “dynamic UI updates” means in Wt

In Wt, the browser is a rendering endpoint, not the primary application runtime. The authoritative state of the UI lives on the server as a widget tree owned by the session’s `WApplication`. A *dynamic UI update* is therefore:

- a server-side change to widget state (text, visibility, classes, models),

- triggered by an event (click, input, timer, push),
- synchronized to the browser as incremental DOM updates.

This model differs from SPA frameworks where the UI is computed primarily in the browser and synchronized from client-side state. In Wt, you update widgets in C++ and Wt updates the browser.

8.3.2 The fundamental update pipeline

Every dynamic update follows the same pipeline:

1. user action or server signal occurs,
2. Wt dispatches an event into the session,
3. your handler modifies the widget tree/state,
4. Wt computes the minimal set of changes,
5. those changes are sent to the browser and applied.

A professional UI design attempts to keep updates:

- localized (update a small region, not the full page),
- predictable (consistent transitions and states),
- bounded (avoid generating enormous DOM diffs).

8.3.3 Immediate updates from event handlers

The simplest and most common dynamic updates happen directly in event handlers.

Text and status updates

```
#include <Wt/WText.h>
#include <Wt/WPushButton.h>

auto status = root()->addWidget(std::make_unique<Wt::WText>("Idle"));
auto btn = root()->addWidget(std::make_unique<Wt::WPushButton>("Run"));

btn->clicked().connect([=] {
  status->setText("Running...");
});
```

Visibility toggles

```
#include <Wt/WContainerWidget.h>

auto panel = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
panel->addWidget(std::make_unique<Wt::WText>("Advanced settings"));
panel->hide();

auto toggle = root()->addWidget(std::make_unique<Wt::WPushButton>("Toggle"));
toggle->clicked().connect([=] {
  if (panel->isHidden()) panel->show();
  else panel->hide();
});
```

CSS class updates

Use CSS classes as your primary mechanism for styling transitions and state:

```
auto box = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
```

```
box->setStyleClass("card");

auto highlight = root()->addWidget(std::make_unique<Wt::WPushButton>("Highlight"));
highlight->clicked().connect([=] {
  box->toggleStyleClass("card-highlight");
});
```

This keeps presentation in CSS and avoids mixing styling logic into C++.

8.3.4 Updating subtrees: region replacement vs incremental change

Production systems usually apply one of two patterns:

Pattern A: incremental updates within a stable subtree

Preferred when only a small part changes, such as a counter or status label.

- minimal server work,
- minimal client DOM diff,
- best performance.

Pattern B: replace a region subtree (content swapping)

Preferred for navigation or major view transitions (e.g., switching pages in the content area) while keeping the shell stable.

```
void replaceContent(Wt::WContainerWidget& region,
                    std::unique_ptr<Wt::WWidget> view)
{
  region.clear();
```

```
  region.addWidget(std::move(view));
}
```

This is the foundation of professional shells: header and sidebar persist; content is swapped.

8.3.5 Dynamic list/table updates with models

For lists and tables, professional Wt applications avoid manually rebuilding large widget trees. Instead, they use model-driven widgets and update the model.
A conceptual pattern:

- UI uses a model-backed view (table/list),
- updates occur by changing the underlying model data,
- the view reflects changes efficiently.

This keeps large data updates manageable and reduces UI coupling.

8.3.6 Timers: periodic dynamic updates

Dashboards, monitoring screens, and admin consoles often refresh periodically. `WTimer` is the standard session-scoped tool.

```
#include <Wt/WTimer.h>
#include <Wt/WText.h>

auto counterText = root()->addWidget(std::make_unique<Wt::WText>("0"));
int counter = 0;

auto timer = root()->addChild(std::make_unique<Wt::WTimer>());
timer->setInterval(std::chrono::seconds(1));
```

```
timer->timeout().connect([=, &counter] mutable {
  ++counter;
  counterText->setText(std::to_string(counter));
});
timer->start();
```

Production guidance:

- do not refresh expensive data too frequently,
- use server-side caching and incremental updates,
- stop timers when the page/region is not visible or not needed.

8.3.7 Debounced updates (live search and reactive filters)

Some events occur at high frequency (typing). A professional UI debounces to avoid event storms:

```
#include <Wt/WLineEdit.h>
#include <Wt/WTimer.h>

auto search = root()->addWidget(std::make_unique<Wt::WLineEdit>());
search->setPlaceholderText("Search");

auto debounce = root()->addChild(std::make_unique<Wt::WTimer>());
debounce->setInterval(std::chrono::milliseconds(250));

debounce->timeout().connect([=] {
  debounce->stop();
  auto q = search->text().toUTF8();
```

```
  // run filtered query using q, then update UI region
});

search->textInput().connect([=] {
  debounce->stop();
  debounce->start();
});
```

This pattern is critical for scalable production dashboards and admin screens.

8.3.8 Long-running work: asynchronous updates and busy states

Dynamic UI updates become challenging when actions take time. In production:

- do not block the session thread with slow operations,
- always provide immediate feedback (busy indicator),
- re-enable controls only when work is complete,
- update UI safely from the session context.

A minimal busy-state pattern (synchronous placeholder):

```
auto btn = root()->addWidget(std::make_unique<Wt::WPushButton>("Generate report"));
auto status = root()->addWidget(std::make_unique<Wt::WText>());

btn->clicked().connect([=] {
  btn->setDisabled(true);
  status->setText("Working...");
  // In real production code, move heavy work to a worker and update UI on
  ↪  completion.
});
```

The non-negotiable rule:

> Do not update widgets from background threads; marshal results back into the session-safe context.

8.3.9 Server push: updates initiated by the server

Some applications require UI updates that are not triggered by user input:

- notifications,
- live monitoring values,
- job completion events,
- message queues.

Wt supports server-initiated updates (push) within the session model. The production concept is:

1. an external event occurs (job completes),
2. session is notified,
3. the UI region is updated in the session context,
4. changes are pushed to the browser.

When implementing push, maintain strict boundaries:

- never hold raw widget pointers in global structures without lifetime control,
- always verify that the session still exists,
- rate-limit push updates to avoid flooding clients.

8.3.10 Partial rendering strategy: update small regions

Professional systems avoid full page rebuilds for dynamic updates. Instead, define stable regions:

- header region (rarely changes),
- sidebar region (changes on auth/role or active route),
- content region (changes frequently),
- notification region (small, frequently updated),
- modal/dialog layer (dynamic).

Then update only what is relevant. This keeps diffs small and performance stable.

8.3.11 Consistency rules for dynamic UI updates

A production UI must feel consistent. Standardize:

- busy indication (spinner, status line, disabling buttons),
- success/failure notifications (toasts, banners),
- validation errors (field-level + summary),
- navigation updates (content swap rules),
- refresh policies (timers and push frequency).

Consistency is a reliability feature. It reduces user errors and support requests.

8.3.12 Common pitfalls and how to avoid them

- **Event storms:** debounce input-driven updates and avoid heavy work per keystroke.
- **Rebuilding huge trees:** use region updates and model-driven views.
- **Background thread UI mutation:** marshal updates to session-safe execution.
- **Unbounded session memory:** avoid storing large datasets in widgets; paginate and load on demand.
- **Inconsistent feedback:** centralize the notification and error display policy.

8.3.13 Summary

Dynamic UI updates are the core operational strength of Wt. You update server-side widgets in response to events, timers, or push signals, and Wt synchronizes incremental changes to the browser. Production-grade design uses stable layout regions, localized updates, debounced reactive inputs, model-driven views for large data, consistent busy and error feedback patterns, and strict session-safe rules for asynchronous work. With these disciplines, Wt can deliver highly interactive interfaces while preserving server-side correctness and long-term maintainability.

8.4 Page navigation management

8.4.1 Why navigation is an architecture problem in Wt

In classic web frameworks, navigation is mainly URL routing: request arrives, route handler runs, response is rendered. In Wt, the application is **stateful** and the UI is a **persistent server-side widget tree**. This changes the nature of navigation:

- “navigation” usually means swapping the *content region* while keeping the shell alive,
- the URL should still represent state for usability and deep linking,
- browser Back/Forward should behave predictably,
- role-based access should control which pages exist for a user,
- session lifetime must be respected (a page is a session-owned widget).

Therefore, page navigation management in Wt is not just a UI feature. It is an architectural subsystem that coordinates URL state, view composition, authorization, and component lifetimes.

8.4.2 The professional layout assumption: stable shell + replaceable content

A production Wt app should establish a stable shell:

- header (global actions, branding),
- sidebar (navigation),
- content area (pages),
- optional notification area and dialog layer.

Navigation should replace only the content area. This avoids:

- rebuilding header/sidebar repeatedly,
- losing persistent UI state unnecessarily,
- creating large widget diffs for small changes.

8.4.3 The three navigation responsibilities

A professional navigation system usually handles:

1) Route selection

Given a requested path (or menu action), decide which page/view to show.

2) View composition

Instantiate the correct page widget and attach it to the content region, or reuse an existing instance if appropriate.

3) URL and history integration

Keep the browser address bar and Back/Forward behavior consistent with the shown page.

8.4.4 Navigation mechanisms in Wt

Wt provides primitives for representing and responding to navigation state. A production system commonly uses:

- internal path management (route-like behavior within a session),
- stacked views (`WStackedWidget`) for switching without rebuilding,
- explicit content replacement (clear + add new page),
- menu/anchor actions to trigger navigation events.

The best choice depends on scale and whether you need deep linking and history behavior.

8.4.5 Internal path as the canonical navigation state

A robust strategy is:

> Make the internal path the canonical source of navigation state, and render pages based on that path.

Benefits:

- deep linking (open a specific page directly),
- browser Back/Forward support,
- predictable navigation across refreshes (within session rules),
- clean separation between navigation decisions and UI widgets.

8.4.6 A minimal router for production use

The following router:

- listens to path changes,
- checks authorization before rendering,
- swaps the content region,
- avoids exposing page internals to the shell.

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <memory>
#include <string>
```

```
class Router {
public:
  Router(Wt::WApplication& app, Wt::WContainerWidget& content)
    : app_(app), content_(content)
  {
    app_.internalPathChanged().connect(this, &Router::onPathChanged);
  }

  void go(const std::string& path) {
    // Use internal path as navigation state
    app_.setInternalPath(path, true);
  }

  void start() {
    onPathChanged(app_.internalPath());
  }

private:
  void onPathChanged(const std::string& path) {
    // Normalize
    const std::string p = path.empty() ? "/" : path;

    // Authorization gate (placeholder; integrate with your auth system)
    if (!isAllowed(p)) {
      render(makeForbidden());
      return;
    }
```

```
  // Route selection
  if (p == "/" || p == "/dashboard") {
    render(makeDashboard());
  } else if (p == "/settings") {
    render(makeSettings());
  } else {
    render(makeNotFound());
  }
}

bool isAllowed(const std::string& path) const {
  // Production: enforce policy based on session context (roles/tenant)
  // Example: block settings unless authorized
  if (path == "/settings") return true;
  return true;
}

void render(std::unique_ptr<Wt::WWidget> page) {
  content_.clear();
  content_.addWidget(std::move(page));
}

std::unique_ptr<Wt::WWidget> makeDashboard() {
  auto w = std::make_unique<Wt::WContainerWidget>();
  w->addWidget(std::make_unique<Wt::WText>("Dashboard"));
  return w;
}

std::unique_ptr<Wt::WWidget> makeSettings() {
```

```
    auto w = std::make_unique<Wt::WContainerWidget>();
    w->addWidget(std::make_unique<Wt::WText>("Settings"));
    return w;
  }

  std::unique_ptr<Wt::WWidget> makeForbidden() {
    auto w = std::make_unique<Wt::WContainerWidget>();
    w->addWidget(std::make_unique<Wt::WText>("403 Forbidden"));
    return w;
  }

  std::unique_ptr<Wt::WWidget> makeNotFound() {
    auto w = std::make_unique<Wt::WContainerWidget>();
    w->addWidget(std::make_unique<Wt::WText>("404 Not Found"));
    return w;
  }

  Wt::WApplication& app_;
  Wt::WContainerWidget& content_;
};
```

8.4.7 Integrating navigation with header and sidebar

Header buttons and sidebar entries should trigger route changes through the router, not directly rebuild the UI.

```
auto dashboardBtn =
↪   sidebar->addWidget(std::make_unique<Wt::WPushButton>("Dashboard"));
dashboardBtn->clicked().connect([&router] { router.go("/dashboard"); });
```

```
auto settingsBtn =
↪  sidebar->addWidget(std::make_unique<Wt::WPushButton>("Settings"));
settingsBtn->clicked().connect([&router] { router.go("/settings"); });
```

This ensures:

- one source of truth (internal path),
- consistent Back/Forward history,
- consistent authorization enforcement.

8.4.8 Stacked navigation: when it is appropriate

Some applications prefer keeping page instances alive (e.g., dashboards with active filters, multi-step screens). A stacked approach can be effective:

- create all pages once,
- switch using `WStackedWidget`,
- optionally still update internal path to reflect the active page.

This trades memory for faster switching and state retention.

8.4.9 Deep links and parameterized routes

Production applications need parameterized pages such as:

- `/users/42`,
- `/orders/2026-01`,
- `/reports/sales?from=...&to=...`

A professional routing approach:

- parse the internal path into segments,
- validate and convert ids to typed values,
- never use raw segments directly in queries,
- route to a page that loads data through services.

Minimal path parsing:

```
static std::vector<std::string> splitPath(const std::string& p) {
  std::vector<std::string> out;
  std::string cur;
  for (char ch : p) {
    if (ch == '/') {
      if (!cur.empty()) { out.push_back(cur); cur.clear(); }
    } else cur.push_back(ch);
  }
  if (!cur.empty()) out.push_back(cur);
  return out;
}
```

Then match patterns like `["users","42"]` and validate `42`.

8.4.10 Navigation and authorization: do not rely on hidden links

A professional system must enforce access control at routing time:

- if a user is not authorized for a route, render forbidden,
- do not merely hide sidebar entries,

- enforce the same rule in services for data access and mutations.

This is essential because deep links bypass the UI menu layer.

8.4.11 Navigation and session lifetime

Because Wt sessions are stateful:

- a refresh or a new tab may create a new session depending on deployment configuration,
- pages should be able to initialize from the current internal path,
- critical state should live in services or persistence, not only in widgets.

A robust rule:

> If the user re-enters a route, the page should be reconstructible from stable identifiers, not fragile widget state.

8.4.12 Navigation and UX consistency: active state and breadcrumbs

A professional navigation system usually adds:

- active highlighting in the sidebar based on current path,
- breadcrumbs for deep routes,
- title updates based on route.

Active highlighting example:

```
void setActive(Wt::WWidget& item, bool active) {
  if (active) item.addStyleClass("nav-active");
  else item.removeStyleClass("nav-active");
}
```

This provides consistent user orientation across the system.

8.4.13 Error handling routes

Production navigation must be robust against invalid paths and errors:

- 404 route for unknown internal paths,
- 403 route for unauthorized access,
- fallback route for unexpected exceptions (with safe logging).

Never crash the session because a user typed an invalid path.

8.4.14 Recommended directory organization for navigation code

A clean UI architecture usually places navigation in a dedicated module:

```
src/ui/
  layout/
    AppShell.h
    Router.h
  pages/
    DashboardPage.h
    SettingsPage.h
    NotFoundPage.h
    ForbiddenPage.h
```

This prevents navigation logic from polluting `WApplication` and keeps pages modular.

8.4.15 Summary

Page navigation management in Wt is an architectural subsystem that coordinates routing, view composition, URL/history behavior, and authorization in a stateful session-driven environment.

A professional pattern uses a stable shell (header/sidebar/content) and swaps only the content area. The internal path can serve as the canonical navigation state, enabling deep links and consistent Back/Forward behavior. A router listens for path changes, enforces access rules, instantiates the correct page widgets, and renders 403/404 fallbacks safely. With disciplined navigation management, Wt applications remain maintainable, secure, and predictable as they scale.

Part V

The Live Project: User Management System

Designing the User System

9.1 System requirements

This section defines the **practical, production-grade requirements** for building and deploying the live *User Management System* (UMS) using **Wt (WebToolkit)** in Modern C++. The goal is to ensure that the project can be built reproducibly, runs reliably under load, and supports the typical non-functional needs of real user systems: authentication, session management, secure transport, persistence, auditing, backups, and observability.

9.1.1 Target deployment profiles

A user management system can be deployed in more than one shape depending on scale and operational constraints. The same codebase should support these standard profiles:

- **Single-host (simple production)**: one machine running the Wt application server (built-in HTTP(S)/WebSocket connector) and the database on the same host.
- **Reverse-proxy fronted**: a reverse proxy (commonly NGINX or Apache) handles TLS termination and static assets, and forwards requests to the Wt application server.
- **FastCGI deployment**: Wt application runs as a FastCGI service behind a FastCGI-capable web server.

- **Multi-host (scalable)**: separate hosts/containers for web tier and database tier, with optional caching, load balancing, and centralized logging/metrics.

9.1.2 Operating systems and runtime environment

Wt is designed for **Unix-like platforms** and **Windows**. For production, prefer a server-grade OS with long-term maintenance and predictable package management.

- **Linux**: recommended for most deployments (common in hosting and container platforms).
- **BSD/macOS**: viable for development or specialized deployments.
- **Windows Server**: supported for building and hosting depending on your toolchain and operational requirements.

At runtime, the environment must provide:

- **Accurate system time** (NTP/chrony): critical for tokens, sessions, password reset windows, and audit logs.
- **Stable DNS resolution**: for outbound integrations (SMTP, OAuth providers, external APIs).
- **Secure secrets storage**: environment variables, OS-level secrets store, or container secrets (avoid hardcoding).

9.1.3 Compiler and language standard

A production user system benefits from a modern compiler that correctly implements the C++ standard library, concurrency primitives, and security-related toolchain hardening.

- **Language baseline**: C++17 or newer is a practical baseline for modern Wt applications.
- **Recommended**: C++20 for improved correctness and expressiveness (concepts, ranges in supporting libraries, better constexpr facilities).
- **Toolchains**:
  - GCC or Clang on Linux/macOS/BSD.
  - MSVC (Visual Studio) on Windows.

9.1.4 Build system and tooling

Use a CMake-based build to ensure portability and reproducibility.

- **CMake**: required for a clean multi-platform build workflow.
- **Ninja or Make**: as the underlying build executor.
- **CTest**: for unit/integration tests.
- **Sanitizers (dev/test)**: AddressSanitizer/UBSan on Clang/GCC to reduce memory and undefined-behavior risk.

Minimal CMake skeleton for a Wt user system

```
cmake_minimum_required(VERSION 3.20)
project(UserManagementSystem LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

```
find_package(Wt REQUIRED COMPONENTS Wt HTTP Dbo)
find_package(Threads REQUIRED)

add_executable(ums
  src/main.cpp
  src/App.cpp
  src/auth/PasswordPolicy.cpp
  src/auth/LoginService.cpp
  src/db/Schema.cpp
  src/db/UserRepository.cpp
)

target_link_libraries(ums PRIVATE
  Wt::Wt
  Wt::HTTP
  Wt::Dbo
  Threads::Threads
)

# Prefer hardening flags in production builds (tune for your toolchain).
# Use your organization's policy and CI validation for final flags.
```

9.1.5 Core Wt components required by the project

A user management system typically relies on:

- **Core Wt library**: UI layer, widgets, event loop, sessions.
- **HTTP(S)/WebSocket connector**: either
  - **Built-in HTTP(S) connector** (for direct deployments and many production use

cases), or

  - **FastCGI connector** (when integrating tightly with a web server that manages request lifecycle).

- **Database access layer**: **Wt::Dbo** for ORM-like mapping and transactional persistence.

9.1.6 Database requirements

A real user system needs durable storage for:

- user accounts and profiles,
- password hashes / credential metadata,
- roles/permissions,
- sessions (optional: can be in-memory or external cache),
- security events (login attempts, lockouts, password resets),
- audit logs and administrative actions.

Wt::Dbo supports multiple backends commonly used in production:

- **PostgreSQL**: recommended for robustness and operational features.
- **MySQL / MariaDB**: common and well-supported in many hosting environments.
- **SQLite**: excellent for embedded, single-host, or low-ops deployments; also useful for tests.
- **Microsoft SQL Server**: viable for Windows-centric environments.

Database sizing guidance

Sizing depends on concurrency and audit volume. As a conservative baseline:

- **Development**: 1–2 CPU cores, 2–4 GB RAM, SQLite or a small local DB.
- **Small production**: 2–4 cores, 4–8 GB RAM, SSD storage, PostgreSQL/MySQL.
- **Audit-heavy**: prioritize storage IOPS and retention planning (partitioning/archival).

Operational database requirements

- **Backups**: automated, tested restores, and retention policy.
- **Migrations**: controlled schema evolution (versioned migrations).
- **Connection management**: pooling and timeouts aligned with Wt deployment model.
- **Least privilege**: DB user roles with minimal permissions for the app runtime.

9.1.7 Security and cryptography requirements

User management is a security system by nature. The deployment must provide:

Transport security (TLS)

- **HTTPS is mandatory** in production.
- Terminate TLS either:
  - directly in the Wt HTTP(S) connector, or
  - at a reverse proxy in front of Wt.
- **Certificates**: managed via ACME automation or enterprise PKI.

Password storage and credential policy

- **Never store plaintext passwords**.
- Store **salted, slow password hashes** (algorithm choice and parameters must follow current security policy).
- Enforce **rate limiting** and **account lockout** policy for repeated failures.
- Provide **password reset** via time-bound tokens and verified delivery channel (email/SMS depending on scope).

Session and token security

- Use **secure cookies** (Secure, HttpOnly, SameSite) and rotate session identifiers after privilege changes.
- Ensure server-side session invalidation on logout and critical security events.
- Maintain a **server clock discipline** (NTP) to keep token windows reliable.

9.1.8 Web server / connector requirements

Choose one of these connector approaches based on operational needs:

Built-in HTTP(S)/WebSocket connector

- Suitable for development and many production scenarios.
- Supports efficient deployments from small systems up to multi-server environments.
- Simplifies architecture (fewer moving parts) when TLS termination and routing are handled directly or with a thin reverse proxy.

FastCGI deployment

When deploying as FastCGI:

- A **FastCGI-capable web server** is required (for example, Apache, NGINX, or Lighttpd).
- The system must provide the **FastCGI library and C++ bindings** (commonly `libfcgi` and `libfcgi++` on Unix-like systems).

Reverse proxy requirements (recommended in many deployments)

A reverse proxy is strongly recommended when you need:

- centralized TLS termination,
- standardized HTTP security headers,
- request buffering and upload limits,
- DoS mitigations and rate limiting,
- structured access logs.

9.1.9 Email and notification requirements

Most user systems require email for:

- account verification,
- password reset,
- security alerts (new device login, suspicious activity),
- administrative invites.

System requirements:

- Access to an **SMTP relay** (internal mail gateway or external provider).
- Proper DNS records (SPF/DKIM/DMARC) where applicable to reduce spam classification.
- A secure secret store for SMTP credentials.

9.1.10 Storage requirements

- **Persistent storage**: database files, uploaded avatars/documents (if supported), audit exports.
- **Log storage**: application logs, access logs, security event logs.
- **Backup storage**: off-host or off-site recommended; encryption at rest for sensitive environments.

9.1.11 Performance and scalability requirements

User systems are typically constrained by:

- concurrent connections (especially with long-lived sessions and server-initiated updates),
- database query efficiency (login, session validation, role checks),
- cryptographic cost (password hashing and token verification),
- I/O (logs and audit trails).

Recommended baseline configuration decisions:

- Keep authentication endpoints **lightweight** and **rate-limited**.

- Ensure the database has appropriate indexes for `email/username`, `user_id`, `session_id`, and audit time ranges.
- Use a reverse proxy with tuned limits for request bodies and connection handling.

9.1.12 Observability requirements

A production-grade user system must be diagnosable. Requirements include:

- **Structured logging**: timestamps, request correlation IDs, user IDs (where safe), and security event categories.
- **Metrics**: request rate, latency distributions, error rates, authentication failures, lockouts, DB latency.
- **Tracing (optional but valuable)**: end-to-end visibility across proxy, app, and DB.

9.1.13 Testing and quality gates

The system requirements for quality assurance are part of production readiness:

- **Unit tests**: password policy, token generation/validation, permission checks.
- **Integration tests**: database schema, migrations, login flow, session lifecycle.
- **Security tests**: rate limiting, brute-force scenarios, session fixation prevention, input validation.
- **CI pipeline**: reproducible builds, tests, and packaging for target platforms.

A minimal test entry point (illustrative)

```
#include <cassert>
#include <string>

int main() {
  // Example: enforce basic invariants used by your password policy component.
  const std::string pw = "CorrectHorseBatteryStaple!";
  assert(pw.size() >= 12);

  // In the real project, replace this with unit tests using your chosen framework.
  return 0;
}
```

9.1.14 Recommended production hardening checklist

- HTTPS everywhere; secure cookies and strict session lifecycle.
- Database backups with restore drills; encryption where required.
- Rate limiting for login and reset flows; audit of authentication events.
- Least-privilege credentials for DB and SMTP.
- Centralized logs and metrics; alerting on anomalies (spikes in failures, latency, or errors).
- Automated builds and deployments; pinned dependencies; reproducible toolchains.

9.1.15 Summary

A Wt-based User Management System is a full production service: it requires a modern C++ toolchain, a deliberate connector strategy (built-in HTTP(S)/WebSocket or FastCGI), a

supported database backend through Wt::Dbo, and a security-first operational environment (TLS, safe password storage, strict session policy, logging, backups, and monitoring). These system requirements are the foundation on which the rest of Chapter 9 will build the architecture, domain model, persistence, and authentication flows.

9.2 Use cases

This section defines the **use cases** of the live *User Management System* (UMS) implemented with Wt. These use cases drive the domain model (users, credentials, sessions, roles), the UI structure, and the persistence/auditing strategy. A production user system is not only "login and logout"; it is the security boundary for the entire application and must support real operational workflows.

9.2.1 Actors

- **Anonymous visitor**: has no authenticated session, can access public pages and entry points (sign-up, login, password reset).
- **Authenticated user**: a verified identity with a session and a set of permissions.
- **Privileged operator (admin/support)**: manages users, roles, access policies, and incident response (lock/unlock, reset MFA, view audit).
- **External identity provider**: optional actor for SSO/OAuth/OIDC integrations.
- **System services**: email delivery, logging/metrics pipeline, database, and background job runner (if used).

9.2.2 Account lifecycle use cases

UC-01: Sign up (self-registration)

Goal: create a new user identity and initial credentials.
Flow (typical):

- Visitor opens registration page.
- Enters primary identifier (email/username) and password (or SSO).
- System validates input, enforces password policy, and checks uniqueness.
- System persists a new `User` record and a `Credential` record (hashed).
- If email verification is required, system issues a verification token and sends an email.
- User confirms verification; account becomes active.

Key requirements:

- Bot resistance (rate limiting; optional CAPTCHA depending on threat model).
- Secure password hashing and safe storage of verification tokens.
- Consistent error messaging that avoids account enumeration.

UC-02: Invite-based registration (admin-driven onboarding)

Goal: allow controlled enrollment, common in enterprise apps.
Flow:

- Admin creates an invitation with role(s) and optional expiration.
- System emails invite link with a time-bound token.

- Invitee sets password (or binds SSO), completes profile, and accepts terms.
- System activates account and records audit events for the invitation lifecycle.

UC-03: Profile completion and updates

Goal: allow users to manage non-sensitive profile data.
Typical operations:

- Update display name, locale/timezone, avatar (if supported), notification preferences.
- Update email address (requires re-verification).
- Update password (requires current password re-authentication).

9.2.3 Authentication and session use cases

UC-10: Login with password

Goal: establish an authenticated session for an existing account.
Flow:

- User submits identifier and password.
- System validates credentials, applies throttling/rate limiting.
- On success, system creates/rotates session identifier and sets secure cookie attributes.
- System records audit event: login success (with device/IP context if captured).

Failure handling:

- On failure, increment counters and emit audit event: login failure.
- Trigger lockout or step-up verification after repeated failures per policy.

UC-11: Logout

Goal: terminate the authenticated session.
Flow:

- User clicks logout.
- System invalidates the server-side session (and/or token) and clears cookie.
- System records audit event: logout.

UC-12: "Remember me" / persistent login (optional)

Goal: enable longer-lived sessions with explicit user consent.
Requirements:

- Persistent tokens must be revocable and rotated.
- User can review and revoke remembered devices in account settings.
- Additional safeguards for privileged accounts (shorter max age, step-up).

UC-13: Session timeout and renewal

Goal: reduce risk from abandoned sessions.

- Idle timeout: session expires after inactivity.
- Absolute timeout: session expires after a maximum lifetime.
- Renewal policy: either re-login or refresh token (for trusted contexts).

UC-14: Step-up authentication for sensitive actions

Goal: require recent authentication for high-risk operations.

Examples:

- Changing password or email address.
- Enabling/disabling MFA.
- Accessing personal data export.
- Performing admin actions (lock/unlock user, role changes).

9.2.4 Password reset and recovery use cases

UC-20: Password reset request

Goal: allow account recovery without revealing whether an account exists.

Flow:

- User submits email/identifier.
- System responds with a generic message and, if applicable, emails a reset link.
- Reset link contains a short-lived, single-use token.

UC-21: Password reset completion

Flow:

- User opens reset link and sets a new password.
- System validates token, enforces password policy, updates credential hash.
- System invalidates existing sessions and remembered devices as policy dictates.
- System records audit event: password reset.

9.2.5 Authorization and role management use cases

UC-30: Role-based access control (RBAC)

Goal: restrict features by role and permission rather than by ad-hoc checks.
Capabilities:

- Define roles (e.g., `User`, `Manager`, `Admin`, `Auditor`).
- Map roles to permissions (e.g., `user.read`, `user.write`, `admin.panel`).
- Assign roles to users (directly or via groups).
- Evaluate permissions at UI rendering time and at request-handling time.

UC-31: Administrative user management

Admin operations:

- Create user (manual), invite user, deactivate user, delete user (if allowed).
- Lock/unlock account, force logout, revoke sessions.
- Reset password or initiate password reset.
- Assign/revoke roles and group memberships.

UC-32: Audit and compliance views

Goal: provide traceability for security-relevant actions.
Audit events commonly required:

- login success/failure, logout, password change/reset,
- role changes, account lock/unlock, profile changes,

- email verification events, invitation lifecycle,
- administrative actions and access to sensitive views.

9.2.6 Multi-factor authentication (optional but common)

UC-40: Enable MFA

Flow:

- User performs step-up authentication.
- System enrolls a second factor (e.g., TOTP) and issues recovery codes.
- System confirms enrollment and records audit event.

UC-41: MFA challenge on login

Flow:

- After password verification, system prompts for second factor.
- On success, session is established with an "MFA verified" claim.
- On failure, apply throttling and record audit events.

9.2.7 Single Sign-On and external identity (optional)

UC-50: Login via external provider

Goal: allow authentication via corporate or third-party identity.
Typical needs:

- Link external identity to a local user record.

- Support first-time login (auto-provisioning) based on policy.
- Enforce local authorization even when authentication is external.
- Provide admin controls to unlink/disable external identities.

9.2.8 Operational and support use cases

UC-60: Account support and incident response

Support workflows:

- Investigate suspicious login patterns via audit views.
- Temporarily lock account, force logout, revoke remembered devices.
- Require password reset and MFA re-enrollment after compromise.

UC-61: Data export and privacy controls (scope-dependent)

If the product scope requires privacy workflows:

- Export a user profile snapshot (download).
- Request account deletion (soft-delete with retention rules).
- Consent and notification preferences.

9.2.9 Wt-specific interaction patterns for use cases

Because Wt is server-side and event-driven, the same use cases map naturally to:

- **Stateful UI sessions**: the UI reflects authentication state immediately without full page reloads.

- **Server-side validation**: input checks and policy enforcement are centralized and consistent.
- **Fine-grained UI authorization**: widgets and routes can be enabled/disabled based on permissions.
- **Safe navigation**: redirect unauthenticated users to login and then back to the original target.

Illustrative Wt routing pattern: protected pages

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <Wt/WStackedWidget.h>

class App : public Wt::WApplication {
public:
  App(const Wt::WEnvironment& env)
    : Wt::WApplication(env),
      content_(root()->addWidget(std::make_unique<Wt::WStackedWidget>()))
  {
    setTitle("UMS");
    internalPathChanged().connect(this, &App::onPathChange);
    onPathChange(internalPath());
  }

private:
  Wt::WStackedWidget* content_{nullptr};
```

```
bool isAuthenticated() const {
  // Replace with your real auth/session check:
  // e.g., session_.login().loggedIn() or a custom claim in server session.
  return false;
}

void showLogin() {
  auto page = std::make_unique<Wt::WContainerWidget>();
  page->addWidget(std::make_unique<Wt::WText>("Login page (placeholder)"));
  content_->setCurrentWidget(content_->addWidget(std::move(page)));
}

void showDashboard() {
  auto page = std::make_unique<Wt::WContainerWidget>();
  page->addWidget(std::make_unique<Wt::WText>("Dashboard (protected)"));
  content_->setCurrentWidget(content_->addWidget(std::move(page)));
}

void onPathChange(const std::string& path) {
  if (path == "/dashboard") {
    if (!isAuthenticated()) {
      setInternalPath("/login", true);
      return;
    }
    showDashboard();
    return;
  }
  if (path == "/login") {
    showLogin();
```

```
      return;
    }
    setInternalPath("/login", true);
  }
};
```

Illustrative authorization check: permissions gate

```
#include <string>
#include <unordered_set>

class PermissionSet {
public:
  bool has(const std::string& p) const { return perms_.count(p) != 0; }
  void grant(std::string p) { perms_.insert(std::move(p)); }
private:
  std::unordered_set<std::string> perms_;
};

static void requirePermission(const PermissionSet& perms, const std::string&
↪  required) {
  if (!perms.has(required)) {
    // In production, map this to a user-friendly forbidden view and an audit event.
    throw std::runtime_error("Forbidden");
  }
}
```

9.2.10 Acceptance criteria per use case (production baseline)

For the live project, each use case is considered complete only if:

- it is covered by UI flow and server-side enforcement (not UI-only),
- failures are handled without leaking sensitive information,
- critical events generate audit records,
- the database model supports the workflow without inconsistent states,
- the behavior is testable via unit/integration tests.

9.2.11 Summary

The UMS use cases define the real boundaries of the system: account lifecycle, login/session lifecycle, recovery, authorization, administration, auditing, and optional advanced capabilities such as MFA and SSO. In Wt, these use cases naturally align with stateful server-side UI, centralized policy enforcement, and consistent navigation/route protection. The next sections will map these use cases to the domain model, database schema, and UI architecture of the live project.

9.3 User, role, and permission modeling

A production user system is only as strong as its identity and authorization model. In Wt applications, authentication and authorization are typically implemented *above* the core Wt primitives, using a combination of:

- a **user identity** stored in your database (the application's `User` entity),
- an **authentication state** in the running session (logged-in or not),
- and an **authorization policy** (roles/permissions, plus record-level ownership checks).

This section presents a robust **RBAC** (Role-Based Access Control) model with **permission granularity** suitable for modern systems: least privilege, auditable changes, controlled escalation, and predictable evaluation performance.

9.3.1 Design goals and constraints

The model must satisfy these constraints:

- **Stable identity**: a user has a durable, non-reusable primary key; external identifiers (email) may change.
- **Separation of authentication and authorization**: "can prove who you are" is not "can do X".
- **Least privilege**: default user capabilities are minimal; privileges are additive and explicitly granted.
- **No client trust**: permissions are enforced server-side in every privileged action (not only in UI visibility).
- **Auditability**: any change to roles/permissions assignments is recorded with actor/time/context.
- **Performance**: permission checks are frequent; evaluation must be fast and cacheable per session.
- **Extensibility**: support future needs (scoped roles, hierarchical roles, per-resource permissions, SSO claims).

9.3.2 Core concepts and terminology

User

A **user** is the application identity:

- immutable primary key (e.g., integer or UUID),
- login identifier(s) (email/username), verified flags, and timestamps,
- status flags (active, locked, suspended, deleted/soft-deleted),
- profile attributes (display name, locale, timezone) which are not security-critical.

Role

A **role** is a named bundle of permissions, usually mapped to a job function or responsibility:

- examples: `user`, `support`, `admin`, `auditor`,
- roles are assigned to users (many-to-many),
- roles may be global or scoped (tenant/project) depending on your product.

Permission

A **permission** is the smallest authorization unit your system checks. A permission should:

- represent **one capability** (not an entire feature set),
- be **stable and versionable** (do not rename casually),
- be **machine-friendly** for checks and **human-friendly** for administration.

A common naming scheme is `resource.action`:

- `user.read`, `user.write`, `user.disable`,
- `role.assign`, `permission.manage`,
- `audit.read`, `system.settings.write`.

9.3.3 Recommended RBAC model (baseline)

The baseline model uses these relationships:

- **User** $\leftrightarrow$ **Role**: many-to-many (`user_roles`)
- **Role** $\leftrightarrow$ **Permission**: many-to-many (`role_permissions`)

A user inherits the union of permissions from all assigned roles. This is the simplest robust model and fits most products.

When to add direct user permissions

Some systems also allow direct grants/denies per user. This can be useful but increases complexity and risk of "permission drift".
If you need overrides, prefer a controlled extension:

- **user_permissions** for explicit grants (rare),
- **deny rules** only if your policy and UI can explain them clearly.

In many products, **avoid denies** and rely on least privilege + explicit role assignment. Deny rules complicate reasoning, caching, and support workflows.

9.3.4 Entity model (Wt::Dbo)

Wt::Dbo maps C++ classes to relational tables and supports relations using `hasMany()`, `belongsTo()`, and join-table mapping. The following model is designed for:

- a durable `User` entity containing application-level profile and authorization state,
- separate `Role` and `Permission` entities,
- many-to-many collections for assignments.

C++ entities

```
#pragma once
#include <Wt/Dbo/Dbo.h>
#include <Wt/Dbo/Types.h>
#include <Wt/WDateTime.h>

namespace dbo = Wt::Dbo;

class Role;
class Permission;

class User {
public:
  // Identity and profile
  std::string email;              // Unique when verified/active; treat as mutable.
  std::string displayName;
  bool emailVerified {false};

  // Account state
```

```
  bool active {true};
  bool locked {false};
  Wt::WDateTime createdAt;
  Wt::WDateTime updatedAt;
  Wt::WDateTime lastLoginAt;

  // Relations
  dbo::collection< dbo::ptr<Role> > roles;

  template<class Action>
  void persist(Action& a) {
    dbo::field(a, email,         "email");
    dbo::field(a, displayName,   "display_name");
    dbo::field(a, emailVerified, "email_verified");
    dbo::field(a, active,        "active");
    dbo::field(a, locked,        "locked");
    dbo::field(a, createdAt,     "created_at");
    dbo::field(a, updatedAt,     "updated_at");
    dbo::field(a, lastLoginAt,   "last_login_at");

    // Many-to-many: users <-> roles via join table user_roles
    dbo::hasMany(a, roles, dbo::ManyToMany, "user_roles", "user_id", "role_id");
  }
};

class Role {
public:
  std::string name;        // e.g. "admin"
  std::string description; // human readable
```

```
  bool systemRole {false}; // if true, restrict deletion/rename.

  dbo::collection< dbo::ptr<User> > users;
  dbo::collection< dbo::ptr<Permission> > permissions;

  template<class Action>
  void persist(Action& a) {
    dbo::field(a, name,        "name");
    dbo::field(a, description, "description");
    dbo::field(a, systemRole,  "system_role");

    // Inverse side of users <-> roles
    dbo::hasMany(a, users, dbo::ManyToMany, "user_roles", "role_id", "user_id");

    // Many-to-many: roles <-> permissions via join table role_permissions
    dbo::hasMany(a, permissions, dbo::ManyToMany, "role_permissions", "role_id",
    ↪  "permission_id");
  }
};

class Permission {
public:
  std::string key;         // e.g. "user.read"
  std::string description; // human readable
  bool deprecated {false}; // keep old permissions for migration/compat.

  dbo::collection< dbo::ptr<Role> > roles;

  template<class Action>
```

```
  void persist(Action& a) {
    dbo::field(a, key,         "key");
    dbo::field(a, description, "description");
    dbo::field(a, deprecated,  "deprecated");

    // Inverse side of roles <-> permissions
    dbo::hasMany(a, roles, dbo::ManyToMany, "role_permissions", "permission_id",
    ↪  "role_id");
  }
};
```

Session mapping and schema creation

```
#include <Wt/Dbo/Session.h>
#include <Wt/Dbo/backend/Sqlite3.h>
// Or use backend/Postgres, backend/MySQL, backend/MSSQLServer depending on
↪  deployment.

class DbSession {
public:
  dbo::Session session;

  explicit DbSession(const std::string& dbFile) {
    auto sqlite = std::make_unique<dbo::backend::Sqlite3>(dbFile);
    sqlite->setProperty("show-queries", "false");
    session.setConnection(std::move(sqlite));

    mapClasses();
  }
```

```
  void mapClasses() {
    session.mapClass<User>("users");
    session.mapClass<Role>("roles");
    session.mapClass<Permission>("permissions");
  }

  void createSchemaIfMissing() {
    dbo::Transaction tx(session);
    try {
      session.createTables();
    } catch (...) {
      // Tables may already exist. In real systems, use versioned migrations.
    }
    tx.commit();
  }
};
```

9.3.5 Database constraints and invariants

Identifiers and uniqueness

- **Primary keys**: use database-generated integer IDs or UUIDs for `users`, `roles`, `permissions`.
- **Unique role names**: `roles.name` unique.
- **Unique permission keys**: `permissions.key` unique.
- **Unique user login identifier**: enforce uniqueness for the chosen login identifier(s) with careful handling of:
  - verified vs unverified emails,

  - soft-deleted accounts,
  - canonicalization rules (case normalization policy must be consistent).

Join table integrity

For `user_roles` and `role_permissions`:

- enforce a unique composite key (`(user_id, role_id)` and `(role_id, permission_id)`),
- enforce foreign keys with cascade policies aligned to your retention rules,
- ensure changes are transactional and auditable.

9.3.6 Permission evaluation strategy

Authorization checks happen in two places:

- **UI-time**: show/hide widgets, disable buttons, prevent navigation into protected sections.
- **Action-time**: enforce policy in all endpoints, events, and server-side handlers (the real boundary).

Compute effective permissions

Effective permissions are computed as:

$$\textit{EffectivePerms(user)} = \bigcup_{r \in Roles(user)} Perms(r)$$

Optionally union with direct user grants if you implement them.

Cache per session

Because permission checks are frequent, compute once per login/session and cache:

- store a set of permission keys in the Wt application session object,
- invalidate cache when:
  - user roles change,
  - role permissions change,
  - user account is locked/disabled,
  - admin forces logout / revokes sessions.

Minimal permission set representation

Use a **flat set** of strings (or interned IDs) for fast membership checks. For high-scale systems, map permissions to integer IDs and store a bitset-like structure, but keep administration in terms of keys.

```
#include <unordered_set>
#include <string>

class PermissionCache {
public:
  void clear() { perms_.clear(); }
  void add(const std::string& p) { perms_.insert(p); }
  bool has(const std::string& p) const { return perms_.find(p) != perms_.end(); }

private:
  std::unordered_set<std::string> perms_;
};
```

Load effective permissions from the database

```
#include <Wt/Dbo/ptr.h>
#include <Wt/Dbo/Transaction.h>

static PermissionCache loadEffectivePermissions(dbo::Session& s, dbo::ptr<User> u)
↪  {
  PermissionCache cache;

  dbo::Transaction tx(s);

  // Force-load roles and their permissions; Wt::Dbo resolves relations lazily.
  for (const dbo::ptr<Role>& r : u->roles) {
    for (const dbo::ptr<Permission>& p : r->permissions) {
      if (!p->deprecated) {
        cache.add(p->key);
      }
    }
  }

  tx.commit();
  return cache;
}
```

9.3.7 Record-level access control (ownership and scope)

RBAC answers: *"Does the user have capability X?"* Many real actions also require: *"Is the user allowed to do X on **this specific record**?"*

Examples:

- A user can read *their own* profile (`user.read.self`) but not others.

- A support agent can read users in their tenant but not across tenants.
- An admin can change roles, but only within their administrative scope.

A robust design combines:

- **capability check** (RBAC permission),
- **ownership/scope check** (record attribute constraints).

Example: enforce “self” vs “admin” access

```
#include <stdexcept>
#include <cstdint>

struct AuthzContext {
  std::int64_t actorUserId;
  bool isAdmin;
  PermissionCache perms;
};

static void require(bool ok, const char* msg) {
  if (!ok) throw std::runtime_error(msg);
}

static void requireCanViewUser(const AuthzContext& ctx, std::int64_t targetUserId)
↪  {
  if (ctx.actorUserId == targetUserId) {
    require(ctx.perms.has("user.read.self") || ctx.perms.has("user.read"),
            "Forbidden: cannot read own profile");
    return;
  }
```

```
  require(ctx.perms.has("user.read.any") || ctx.perms.has("user.read"),
          "Forbidden: cannot read other users");
}
```

9.3.8 Role design guidelines (practical and maintainable)

Prefer fewer roles, more explicit permissions

A common failure mode is creating many near-duplicate roles. Instead:

- define a small set of **core roles** (user, support, admin, auditor),
- keep permissions explicit so roles are composable and auditable,
- avoid "copy Bob" administration (cloning users) as it causes permission creep.

Role hierarchy (optional)

If you need hierarchy (e.g., `admin` includes `support` includes `user`):

- implement it as **role composition** (admin role contains all needed permissions),
- or explicitly model **role inheritance** (more complex; requires careful cycle prevention).

For many systems, role composition via permissions is simpler and safer.

9.3.9 Permission catalog (recommended baseline for the live project)

Below is a practical starting catalog. Adjust to your product features.

User and authentication

- `session.login` authenticate using supported methods
- `session.logout` terminate session
- `user.read.self` read own profile
- `user.write.self` update own profile
- `user.password.change.self` change own password (with step-up)

Administration

- `user.read.any` read any user
- `user.write.any` update any user profile fields allowed by policy
- `user.disable` deactivate/disable accounts
- `user.lock` lock/unlock accounts
- `user.session.revoke` revoke sessions / force logout
- `role.read` view roles
- `role.write` create/update roles
- `role.assign` assign roles to users
- `permission.read` view permission catalog
- `permission.manage` manage permission catalog (rare; typically system-only)

Audit and compliance

- `audit.read` view audit events
- `audit.export` export audit data (subject to retention policy)

9.3.10 Auditing model (minimal requirements)

Authorization systems must be explainable. At minimum, record:

- who performed a change (actor user id),
- what changed (role assignment, permission assignment),
- target entity (user id / role id),
- time and context (timestamp, request id, optional IP/device fingerprint),
- result (success/failure) for enforcement-relevant actions.

A typical audit entity can be independent of RBAC entities to keep retention and storage policy flexible.

9.3.11 Integration with authentication state in Wt

Wt applications typically maintain a per-session object that represents:

- whether the user is authenticated,
- the current user's database identity,
- the permission cache.

Illustrative session-level auth context

```
#include <optional>
#include <cstdint>

class AuthSessionState {
public:
  void onLogin(std::int64_t userId, PermissionCache perms) {
    userId_ = userId;
    perms_ = std::move(perms);
  }

  void onLogout() {
    userId_.reset();
    perms_.clear();
  }

  bool loggedIn() const { return userId_.has_value(); }
  std::int64_t userId() const { return *userId_; }

  const PermissionCache& permissions() const { return perms_; }

private:
  std::optional<std::int64_t> userId_;
  PermissionCache perms_;
};
```

9.3.12 Operational rules and safety checks

Enforce policy in one place

To avoid scattered ad-hoc checks:

- centralize authorization in a small **Authorization Service**,
- provide helper methods: `requirePermission()`, `requireOwnershipOrPermission()`, etc.

Always verify account state

Even if a session exists, every sensitive action must confirm:

- `active == true`,
- `locked == false`,
- credentials not expired (if your policy uses expiration),
- step-up requirements satisfied for sensitive actions.

Prevent privilege escalation

- Only users with `role.assign` can modify roles.
- Restrict assignment of high-privilege roles (e.g., `admin`) to a smaller permission such as `role.assign.admin`.
- For role/permission changes, require step-up authentication and log the event.

9.3.13 Summary

A production-grade user system model is built from three layers:

- **User identity** (durable `User` entity),
- **Role-based permissions** (RBAC: `User`↔`Role` and `Role`↔`Permission`),
- **Record-level rules** (ownership/scope constraints for individual resources).

In Wt::Dbo, this maps cleanly to many-to-many relations via join tables, and in the application runtime it is evaluated through a cached permission set combined with server-side enforcement. This modeling approach scales from small deployments to multi-operator production systems while remaining auditable, maintainable, and resistant to common authorization failures.

Login Page Implementation

10.1 Building the login view

This section implements the **Login View** for the live User Management System (UMS) using Wt. The login page is not merely a form; it is a security boundary and a user experience gateway. A correct implementation must satisfy:

- **Clean UI layout**: clear structure, predictable focus behavior, accessible labels.
- **Server-side authority**: all checks and authentication decisions happen on the server.
- **Safe error handling**: prevent account enumeration and reduce information leakage.
- **Rate limiting hooks**: defend against brute-force attempts (even if enforced elsewhere).
- **Navigation integration**: redirect authenticated users to the intended protected path.

10.1.1 UI goals and interaction contract

The login view must support the following interaction contract:

- Accept an **identifier** (email or username) and a **password**.
- Provide a **primary action** (Log in) and optional secondary actions:

  - **Remember me** (optional feature; affects session duration policy).
  - **Forgot password** navigation.
  - **Create account** navigation (if self-registration is allowed).

- Validate user input **before** attempting authentication.
- On authentication failure, show a **generic** message.
- On success, create/refresh the server-side session state and redirect.

Because Wt applications are stateful and event-driven, the login view is implemented as a widget tree driven by server-side event handlers. The server remains the source of truth.

10.1.2 Recommended view structure

A production-friendly structure for the login page typically contains:

- **Header**: title and brief guidance.
- **Form panel**: identifier input, password input, remember-me checkbox.
- **Feedback area**: validation and authentication messages.
- **Actions**: Log in button (primary), links for reset and registration (secondary).

Wt supports building this structure using widgets directly, or using `Wt::WTemplate` for clearer separation of layout and logic. The template approach is recommended for maintainability.

10.1.3 Service interfaces used by the view

The login view should not embed persistence or cryptography logic. Instead, it consumes an **authentication service** and a **session state** object.

Authentication service contract

The login view needs a single operation:

- Authenticate by identifier + password.

The service returns a **result** indicating success/failure and, on success, the authenticated user identity.

```
#pragma once
#include <cstdint>
#include <string>

struct AuthResult {
  bool ok{false};
  std::int64_t userId{0};     // valid only when ok == true
  bool locked{false};         // optional: indicate locked state internally
  bool disabled{false};       // optional: indicate disabled state internally
};

class IAuthService {
public:
  virtual ~IAuthService() = default;

  // Authenticate by identifier (email/username) and password.
  // The service enforces password hashing verification and policy.
  virtual AuthResult authenticate(const std::string& identifier,
                                  const std::string& password) = 0;
};
```

Session state contract

The view needs to set a logged-in state and permission cache (or a minimal identity claim).

```
#pragma once
#include <cstdint>
#include <optional>
#include <string>
#include <unordered_set>

class PermissionCache {
public:
  void clear() { perms_.clear(); }
  void add(const std::string& p) { perms_.insert(p); }
  bool has(const std::string& p) const { return perms_.count(p) != 0; }
private:
  std::unordered_set<std::string> perms_;
};

class AuthSessionState {
public:
  void onLogin(std::int64_t userId, PermissionCache perms) {
    userId_ = userId;
    perms_ = std::move(perms);
  }
  void onLogout() {
    userId_.reset();
    perms_.clear();
  }
  bool loggedIn() const { return userId_.has_value(); }
```

```
  std::int64_t userId() const { return *userId_; }
  const PermissionCache& permissions() const { return perms_; }

private:
  std::optional<std::int64_t> userId_;
  PermissionCache perms_;
};
```

10.1.4 Template-based login view implementation

We now implement a reusable `LoginView` widget. It:

- creates inputs and binds them into a template,
- performs input validation,
- calls `IAuthService::authenticate()`,
- updates `AuthSessionState` on success,
- redirects to the post-login target path.

HTML template (inline string)

Using `Wt::WTemplate` allows a structured layout while keeping logic in C++.

```
static const char* kLoginTemplate = R"(
<div class="auth-page">
  <div class="auth-card">
    <h2>${title}</h2>
    <p class="muted">${subtitle}</p>
```

```
    <div class="auth-feedback">${feedback}</div>

    <div class="form-row">
      <label for="id">${idLabel}</label>
      <div>${identifier}</div>
    </div>

    <div class="form-row">
      <label for="pw">${pwLabel}</label>
      <div>${password}</div>
    </div>

    <div class="form-row form-row-inline">
      <div>${remember}</div>
      <div class="muted">${rememberLabel}</div>
    </div>

    <div class="actions">
      ${loginButton}
      <span class="spacer"></span>
      ${forgotLink}
      ${signupLink}
    </div>
  </div>
</div>
)";
```

LoginView widget

```
#pragma once
```

```
#include <memory>
#include <string>

#include <Wt/WCheckBox.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
#include <Wt/WTemplate.h>
#include <Wt/WAnchor.h>
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WValidator.h>

#include "IAuthService.h"
#include "AuthSessionState.h"

// Optional: provide a callback to compute permissions after login.
// In simple systems you may load roles/permissions immediately in the auth service.
using PermissionLoader = std::function<PermissionCache(std::int64_t userId)>;

class LoginView final : public Wt::WContainerWidget {
public:
  LoginView(IAuthService& auth,
            AuthSessionState& sessionState,
            PermissionLoader loadPerms,
            std::string postLoginPath = "/dashboard",
            std::string forgotPath = "/forgot",
            std::string signupPath = "/signup")
```

```
    : auth_(auth),
      sessionState_(sessionState),
      loadPerms_(std::move(loadPerms)),
      postLoginPath_(std::move(postLoginPath)),
      forgotPath_(std::move(forgotPath)),
      signupPath_(std::move(signupPath))
  {
    buildUi();
    wireEvents();
  }

private:
  IAuthService& auth_;
  AuthSessionState& sessionState_;
  PermissionLoader loadPerms_;

  std::string postLoginPath_;
  std::string forgotPath_;
  std::string signupPath_;

  Wt::WTemplate* tpl_{nullptr};
  Wt::WText* feedback_{nullptr};
  Wt::WLineEdit* identifier_{nullptr};
  Wt::WLineEdit* password_{nullptr};
  Wt::WCheckBox* remember_{nullptr};
  Wt::WPushButton* loginBtn_{nullptr};
  Wt::WAnchor* forgot_{nullptr};
  Wt::WAnchor* signup_{nullptr};
```

```
void buildUi() {
  tpl_ =
  ↪  addWidget(std::make_unique<Wt::WTemplate>(Wt::WString::fromUTF8(kLoginTemplate)));

  tpl_->bindString("title", "Sign in");
  tpl_->bindString("subtitle", "Enter your credentials to continue.");
  tpl_->bindString("idLabel", "Email or username");
  tpl_->bindString("pwLabel", "Password");
  tpl_->bindString("rememberLabel", "Keep me signed in on this device");

  feedback_ = tpl_->bindWidget("feedback", std::make_unique<Wt::WText>(""));

  identifier_ = tpl_->bindWidget("identifier",
  ↪  std::make_unique<Wt::WLineEdit>());
  identifier_->setId("id");
  identifier_->setPlaceholderText("name@example.com");
  identifier_->setAutoComplete(true);

  password_ = tpl_->bindWidget("password", std::make_unique<Wt::WLineEdit>());
  password_->setId("pw");
  password_->setEchoMode(Wt::EchoMode::Password);
  password_->setPlaceholderText("");
  password_->setAutoComplete(false);

  remember_ = tpl_->bindWidget("remember", std::make_unique<Wt::WCheckBox>());

  loginBtn_ = tpl_->bindWidget("loginButton",
  ↪  std::make_unique<Wt::WPushButton>("Log in"));
```

```
  forgot_ = tpl_->bindWidget("forgotLink", std::make_unique<Wt::WAnchor>());
  forgot_->setText("Forgot password?");
  forgot_->setLink(Wt::WLink(Wt::LinkType::InternalPath, forgotPath_));

  signup_ = tpl_->bindWidget("signupLink", std::make_unique<Wt::WAnchor>());
  signup_->setText("Create account");
  signup_->setLink(Wt::WLink(Wt::LinkType::InternalPath, signupPath_));

  // Basic validators (server-side). Keep them conservative and consistent.
  // Identifier: 3..254 characters (email-like max) - do not enforce strict email
  ↪  format
  // if your system also supports usernames.
  auto idv = std::make_shared<Wt::WLengthValidator>(3, 254);
  idv->setMandatory(true);
  identifier_->setValidator(idv);

  // Password: enforce minimal length at UI layer; deeper policy should be in
  ↪  auth/account service.
  auto pwv = std::make_shared<Wt::WLengthValidator>(8, 256);
  pwv->setMandatory(true);
  password_->setValidator(pwv);

  setStyleClass("auth-root");
  feedback_->setStyleClass("auth-feedback-text");
  loginBtn_->setStyleClass("primary");
}

void wireEvents() {
  // Clicking the button triggers login attempt.
```

```
  loginBtn_->clicked().connect(this, &LoginView::attemptLogin);

  // Enter in either field triggers login attempt.
  identifier_->enterPressed().connect(this, &LoginView::attemptLogin);
  password_->enterPressed().connect(this, &LoginView::attemptLogin);

  // Focus the identifier field by default.
  identifier_->setFocus(true);
}

void setFeedback(const std::string& msg, bool isError) {
  feedback_->setText(Wt::WString::fromUTF8(msg));
  feedback_->setStyleClass(isError ? "auth-feedback-text error" :
  ↪  "auth-feedback-text ok");
}

bool validateInputs() {
  // Force validation. Wt validators run server-side as part of events.
  const bool idOk = (identifier_->validate() == Wt::ValidationState::Valid);
  const bool pwOk = (password_->validate() == Wt::ValidationState::Valid);

  if (!idOk || !pwOk) {
    setFeedback("Please provide a valid identifier and password.", true);
    return false;
  }
  return true;
}

std::string computePostLoginTarget() const {
```

```
  // Support deep-linking: if user was redirected to /login?next=/x, store it in
  ↪ internal path.
  // In many Wt apps, you can store the original target in session state or in the
  ↪ internal path.
  // For simplicity, use a fixed default.
  return postLoginPath_;
}

void attemptLogin() {
  // Prevent double-submission.
  loginBtn_->setEnabled(false);

  if (!validateInputs()) {
    loginBtn_->setEnabled(true);
    return;
  }

  const std::string id = identifier_->text().toUTF8();
  const std::string pw = password_->text().toUTF8();

  // Always use generic failure messages to avoid leaking account existence.
  // Details (locked/disabled) may be used for auditing, not for UI disclosure.
  AuthResult r{};
  try {
    r = auth_.authenticate(id, pw);
  } catch (...) {
    setFeedback("Login failed. Please try again.", true);
    loginBtn_->setEnabled(true);
    return;
```

```
  }

  if (!r.ok) {
    // Optional: you may apply exponential backoff or integrate with a rate
    ↪  limiter here.
    setFeedback("Login failed. Check your credentials and try again.", true);
    loginBtn_->setEnabled(true);
    return;
  }

  // On success, load effective permissions and update session state.
  PermissionCache perms;
  if (loadPerms_) {
    perms = loadPerms_(r.userId);
  }
  sessionState_.onLogin(r.userId, std::move(perms));

  // Remember-me policy is not implemented here; it is an application-wide session
  ↪  policy decision.
  // You can store the user's preference and adjust session duration on the server
  ↪  if supported.
  const bool rememberMe = remember_->isChecked();
  (void)rememberMe;

  setFeedback("Signed in successfully.", false);

  // Redirect to post-login target.
  auto* app = Wt::WApplication::instance();
  app->setInternalPath(computePostLoginTarget(), true);
```

```
  }
};
```

10.1.5 Integration: placing the login view in the application router

A production Wt application typically uses **internal paths** for navigation and routing. The router should:

- send unauthenticated users to `/login`,
- after successful login, send them to the intended protected page.

```
#include <Wt/WApplication.h>
#include <Wt/WStackedWidget.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>

class App : public Wt::WApplication {
public:
  App(const Wt::WEnvironment& env,
      IAuthService& auth,
      AuthSessionState& authState,
      PermissionLoader loadPerms)
    : Wt::WApplication(env),
      auth_(auth),
      authState_(authState),
      loadPerms_(std::move(loadPerms))
  {
    setTitle("User Management System");
    stack_ = root()->addWidget(std::make_unique<Wt::WStackedWidget>());
```

```
    internalPathChanged().connect(this, &App::route);
    route(internalPath());
  }

private:
  IAuthService& auth_;
  AuthSessionState& authState_;
  PermissionLoader loadPerms_;
  Wt::WStackedWidget* stack_{nullptr};

  void showLogin() {
    auto page = std::make_unique<Wt::WContainerWidget>();
    page->addWidget(std::make_unique<LoginView>(auth_, authState_, loadPerms_,
                                                "/dashboard", "/forgot", "/signup"));
    stack_->clear();
    stack_->addWidget(std::move(page));
  }

  void showDashboard() {
    auto page = std::make_unique<Wt::WContainerWidget>();
    page->addWidget(std::make_unique<Wt::WText>("Dashboard (protected)"));
    stack_->clear();
    stack_->addWidget(std::move(page));
  }

  void route(const std::string& path) {
    if (path == "/dashboard") {
      if (!authState_.loggedIn()) {
        setInternalPath("/login", true);
```

```
        return;
      }
      showDashboard();
      return;
    }
    if (path == "/login") {
      showLogin();
      return;
    }
    // Default route:
    setInternalPath("/login", true);
  }
};
```

10.1.6 Security requirements embedded in the login view

Generic failure messages

The UI must not reveal:

- whether an identifier exists,
- whether the password is wrong vs account locked,
- whether the account is disabled.

For operational needs, log details server-side (audit), but expose only a generic message to the user.

Rate limiting hooks

The login page is a brute-force target. Even if rate limiting is enforced at a proxy, the application should still support:

- throttling based on identifier and/or client IP,
- exponential backoff after repeated failures,
- account lockout policies (applied carefully to avoid denial-of-service vectors).

In this chapter, the view is designed so rate limiting can be integrated in `IAuthService::authenticate()` or in a separate policy component called by `attemptLogin()`.

Session fixation prevention

After a successful login, ensure that the server-side session identifier is rotated or otherwise protected by your session implementation policy. In Wt-based systems, session behavior is typically managed by the application and connector configuration; ensure the authenticated session cannot be reused from a prior unauthenticated context.

10.1.7 UX improvements for production

- **Disable the login button during submission** (already done) to prevent double-clicking.
- **Keyboard-first behavior**: Enter triggers submit.
- **Focus management**: focus the identifier field initially.
- **Consistent validation messages**: do not show different messages for different failure modes.
- **Accessibility**: link labels with input IDs (`for="id"` and `setId("id")`).

10.1.8 Minimal styling (optional but practical)

The following CSS is intentionally small and can be expanded later. It aims for a centered card layout with a readable form.

```
.auth-page {
  display: flex;
  justify-content: center;
  padding: 2rem 1rem;
}

.auth-card {
  width: 100%;
  max-width: 520px;
  border: 1px solid #ddd;
  border-radius: 10px;
  padding: 1.25rem 1.25rem 1rem 1.25rem;
  background: #fff;
}

.form-row { margin-top: 0.9rem; }
.form-row label { display: block; font-weight: 600; margin-bottom: 0.35rem; }

.form-row-inline {
  display: flex;
  gap: 0.6rem;
  align-items: center;
}

.actions {
```

```
  display: flex;
  align-items: center;
  gap: 0.75rem;
  margin-top: 1.1rem;
}

.actions .spacer { flex: 1; }

.auth-feedback-text { margin-top: 0.6rem; min-height: 1.2rem; }
.auth-feedback-text.error { color: #b00020; }
.auth-feedback-text.ok { color: #0a7a2f; }

.muted { color: #666; }
.primary { font-weight: 700; }
```

10.1.9 Summary

The login view is a stateful Wt widget composed of a template, server-side validators, and secure event handlers. It delegates authentication to an `IAuthService`, updates a session state object on success, and uses internal-path routing to navigate users into protected areas. The design intentionally keeps security decisions server-side, uses generic failure messages, and provides clear integration points for rate limiting, auditing, and session policy hardening.

10.2 Input validation

Input validation in a login page is a **security and reliability mechanism**, not a cosmetic feature. The login endpoint is a high-value target for:

- brute-force password attempts,

- credential stuffing,
- account enumeration,
- injection into downstream layers (logs, templates, SQL, email),
- denial-of-service via oversized payloads or pathological inputs.

In Wt, validation must be treated as a **server-side responsibility** because the UI is rendered by the server and events are processed on the server. Client-side checks (if present) can improve UX, but they must never be the enforcement boundary. This section defines a production-grade validation strategy for the login view.

10.2.1 Validation objectives

A correct validation plan for login forms must meet the following objectives:

- **Reject invalid shapes early**: empty fields, too short/long values, disallowed characters.
- **Protect downstream components**: databases, logs, templates, and rate limiters should never see unbounded or malformed data.
- **Avoid account enumeration**: validation and error responses must not reveal whether the identifier exists.
- **Normalize consistently**: if you canonicalize identifiers (case folding, trimming), do it in one place with clear rules.
- **Be deterministic**: same inputs yield same validation results independent of locale or encoding confusion.
- **Be testable**: validators and normalization rules must have unit tests.

10.2.2 Two-layer validation model

Use two layers:

Layer 1: UI-level (shape validation)

This layer runs at the login view boundary:

- Mandatory fields
- Length constraints
- Basic permitted character set for usernames (if supported)
- “Looks like email” optional check (but do not require it if usernames are allowed)

Purpose:

- prevent accidental errors,
- reduce server work on obviously invalid inputs,
- keep feedback consistent without leaking account existence.

Layer 2: Service-level (security validation)

This layer runs inside the authentication service:

- rate limiting / throttling decision,
- constant-time password hash verification strategy,
- account state checks (locked/disabled),
- audit/event emission.

Important: the authentication service must treat the input as untrusted even if UI validation succeeded.

10.2.3 What to validate on a login form

Identifier field

An identifier can be:

- an email address, or
- a username, or
- both (configurable).

Recommended validation rules:

- Trim surrounding whitespace.
- Enforce minimum length (e.g., 3).
- Enforce maximum length:
  - email-like identifiers: 254 characters is a practical max for many systems,
  - usernames: define a stricter max (e.g., 32–64) if you do not support email login.
- Reject control characters and newlines (protect logs and headers).
- Define allowed characters for usernames (e.g., `[A-Za-z0-9_-.]`), but allow broader input for emails if you accept them.

Normalization rules (must be consistent across registration, login, and account lookup):

- Emails: many systems normalize by trimming and lowercasing the domain part; some lower-case the whole email for simplicity.
- Usernames: define whether case-sensitive or case-insensitive.
- Never perform ambiguous Unicode normalization unless you have a clear policy; mixing scripts and lookalike characters creates security risks.

Password field

A login form should validate only the **shape**:

- Mandatory non-empty.
- Minimum length is optional at login time (because existing users may have older passwords), but you should still cap maximum length to protect CPU cost and memory use.
- Maximum length recommended (e.g., 256 or 1024 depending on policy).

At login time, avoid strict password complexity rules in the UI; those belong to **password creation** and **password change** flows, not authentication. The login form's job is to safely accept input and pass it to the authentication service without allowing abuse.

10.2.4 Wt validators: production usage pattern

Wt provides `Wt::WValidator` and concrete validators such as `Wt::WLengthValidator`. The correct pattern is:

- attach validators to input widgets,
- call `validate()` in the login handler,
- display generic messages on failure,
- keep detailed diagnostics out of the UI.

Reusable validation helpers

```
#pragma once
#include <string>
```

```
#include <algorithm>

static inline std::string trimAsciiWhitespace(std::string s) {
  auto isWs = [](unsigned char c) {
    return c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\f' || c == '\v';
  };

  while (!s.empty() && isWs(static_cast<unsigned char>(s.front())))
  ↪  s.erase(s.begin());
  while (!s.empty() && isWs(static_cast<unsigned char>(s.back())))  s.pop_back();
  return s;
}

static inline bool containsControlChar(const std::string& s) {
  for (unsigned char c : s) {
    if (c < 0x20 || c == 0x7F) return true; // ASCII control chars + DEL
  }
  return false;
}
```

Custom Wt validator for identifiers

Wt's `WRegExpValidator` can be used for strict username patterns, but if you support both email and username, it is often clearer to implement a custom validator that checks:

- presence and length,
- no control characters,
- optional: username pattern if it does not look like email.

```
#pragma once
```

```
#include <Wt/WValidator.h>
#include <Wt/WString.h>
#include <cctype>

class IdentifierValidator final : public Wt::WValidator {
public:
  IdentifierValidator(int minLen, int maxLen, bool allowUsernames)
    : minLen_(minLen), maxLen_(maxLen), allowUsernames_(allowUsernames) {}

  Wt::Result validate(const Wt::WString& input) const override {
    const std::string raw = input.toUTF8();
    const std::string s = trimAsciiWhitespace(raw);

    if (s.empty()) {
      return Wt::Result(Wt::ValidationState::Invalid, "Required");
    }
    if (static_cast<int>(s.size()) < minLen_ || static_cast<int>(s.size()) >
    ↪  maxLen_) {
      return Wt::Result(Wt::ValidationState::Invalid, "Invalid");
    }
    if (containsControlChar(s)) {
      return Wt::Result(Wt::ValidationState::Invalid, "Invalid");
    }

    // If it looks like an email (contains '@'), accept a relaxed shape:
    // Do not implement a complex RFC-style email parser here.
    const bool looksEmail = (s.find('@') != std::string::npos);
    if (looksEmail) {
      // Basic additional checks that are safe and non-opinionated:
```

```
      if (s.front() == '@' || s.back() == '@') {
        return Wt::Result(Wt::ValidationState::Invalid, "Invalid");
      }
      return Wt::Result(Wt::ValidationState::Valid);
    }

    // Otherwise treat as username if allowed.
    if (!allowUsernames_) {
      return Wt::Result(Wt::ValidationState::Invalid, "Invalid");
    }

    // Username policy: ASCII letters/digits plus _.- (adjust as policy)
    for (unsigned char c : s) {
      const bool ok =
        (std::isalnum(c) != 0) || (c == '_') || (c == '-') || (c == '.');
      if (!ok) {
        return Wt::Result(Wt::ValidationState::Invalid, "Invalid");
      }
    }

    return Wt::Result(Wt::ValidationState::Valid);
  }

private:
  int minLen_;
  int maxLen_;
  bool allowUsernames_;
};
```

Login view wiring: attach validators and validate

```
#include <Wt/WLineEdit.h>
#include <Wt/WLengthValidator.h>

static void configureLoginValidators(Wt::WLineEdit* identifier,
                                     Wt::WLineEdit* password) {
  // Identifier: 3..254, allow usernames in addition to email.
  auto idv = std::make_shared<IdentifierValidator>(3, 254, true);
  idv->setMandatory(true);
  identifier->setValidator(idv);

  // Password: accept a safe range to prevent abuse; keep policy deeper in the
  ↪  service.
  auto pwv = std::make_shared<Wt::WLengthValidator>(1, 256);
  pwv->setMandatory(true);
  password->setValidator(pwv);
}

static bool validateLoginForm(Wt::WLineEdit* identifier,
                              Wt::WLineEdit* password) {
  const bool idOk = (identifier->validate() == Wt::ValidationState::Valid);
  const bool pwOk = (password->validate() == Wt::ValidationState::Valid);
  return idOk && pwOk;
}
```

10.2.5 Safe feedback: what to show the user

When validation fails, **do not** reveal fine-grained reasons that help attackers. Use a unified message such as:

- “Please enter a valid identifier and password.”

Reserve detailed feedback for:

- registration and password creation flows,
- authenticated settings screens (after step-up),
- server logs and audit events.

10.2.6 Avoiding common validation pitfalls

Do not trust client-side constraints

Even if you add HTML attributes or JavaScript checks, never rely on them. In Wt, input is processed on the server; treat UI checks as **UX enhancements only**.

Do not use strict email regexes in login

Email standards are complex. A strict regex tends to reject legitimate addresses and break user experience. Prefer:

- a relaxed check on login,
- strict uniqueness and verified-ownership checks at registration and change-email flows.

Cap maximum sizes

Always enforce max sizes on:

- identifier,
- password,
- any hidden fields (CSRF tokens, “next” path),

- headers that your proxy passes through (if your app reads them).

This prevents denial-of-service and resource abuse.

Normalize once and consistently

If you trim or lowercase identifiers:

- do it before database lookup,
- store canonical form in the database (or store both raw and canonical),
- apply the same rule everywhere (registration, login, reset).

10.2.7 Validation inside the authentication service

The authentication service must add security validations that cannot belong to UI:

- **rate limiting**: based on identifier and client context,
- **uniform timing**: avoid early-exit timing that helps enumeration,
- **account state**: locked/disabled checks,
- **audit**: record attempts and results.

A robust pattern is:

- validate shape again (defense in depth),
- apply throttling decision,
- do a consistent password verification path,
- return generic failure to UI while logging details internally.

Illustrative service-level shape checks

```
#include <string>

static bool isValidIdentifierShape(const std::string& id) {
  const std::string s = trimAsciiWhitespace(id);
  if (s.size() < 3 || s.size() > 254) return false;
  if (containsControlChar(s)) return false;
  return true; // deeper checks are policy-specific
}

static bool isValidPasswordShape(const std::string& pw) {
  // Accept any non-empty password but cap to prevent abuse.
  if (pw.empty()) return false;
  if (pw.size() > 256) return false;
  if (containsControlChar(pw)) return false; // optional; prevents log/header issues
  return true;
}
```

10.2.8 Handling redirect targets safely (“next” path)

Many login pages support a post-login redirect target. This parameter is an input and must be validated to avoid:

- open redirects,
- path traversal-like abuse in internal routers,
- injection into templates or logs.

Rules:

- Accept only **internal paths** (e.g., `/dashboard`, `/settings`).
- Reject full URLs and protocol prefixes.
- Enforce a maximum length (e.g., 256).
- Optionally maintain an allow-list of routes.

```
#include <string>

static bool isSafeInternalPath(const std::string& p) {
  if (p.empty() || p.size() > 256) return false;
  if (p[0] != '/') return false;
  if (p.find("://") != std::string::npos) return false;
  if (p.find('\\') != std::string::npos) return false; // keep routing uniform
  if (p.find('\n') != std::string::npos || p.find('\r') != std::string::npos)
  ↪  return false;
  return true;
}
```

10.2.9 Testing the validators

Input validation rules should be covered by unit tests to prevent regressions when the UI evolves.

```
#include <cassert>
#include <string>

int main() {
  assert(isValidIdentifierShape("user@example.com"));
  assert(isValidIdentifierShape("admin"));
```

```
  assert(!isValidIdentifierShape(""));
  assert(!isValidIdentifierShape("a"));
  assert(!isValidIdentifierShape(std::string(300, 'x')));

  assert(isValidPasswordShape("x"));
  assert(!isValidPasswordShape(""));
  assert(!isValidPasswordShape(std::string(300, 'p')));

  assert(isSafeInternalPath("/dashboard"));
  assert(!isSafeInternalPath("https://evil.example/x"));
  assert(!isSafeInternalPath("dashboard"));
  return 0;
}
```

10.2.10 Summary

For the login page, input validation must be **simple, conservative, and security-driven**:

- validate shape and size at the UI boundary using Wt validators,
- normalize identifiers consistently,
- keep error messages generic to reduce enumeration,
- repeat validation in the authentication service (defense in depth),
- validate redirect targets to prevent open redirects,
- cap sizes and reject control characters to protect logs and downstream layers.

This approach yields predictable behavior, safer authentication flows, and a login view that remains robust as the system grows.

10.3 Error handling

Error handling for the login page is a **security control** and an **availability control**. The login endpoint is exposed to untrusted input, high request rates, and deliberate abuse. A production-grade Wt login flow must therefore treat errors as *expected operational events* and handle them in a way that:

- does not leak sensitive information (account enumeration, internal state, stack traces),
- preserves a consistent user experience,
- supports diagnosis through structured logs and audit events,
- remains safe under partial failures (DB down, SMTP down, misconfiguration),
- avoids turning failures into denial-of-service amplifiers (expensive retries, heavy exception paths).

In Wt, UI event handlers run on the server. Any exception or failure inside a handler can affect the current session and user interaction. Therefore, error handling must be designed at two levels:

- **View-level**: translate failures into safe messages and stable UI state.
- **Service-level**: enforce security policy, normalize outcomes, log and audit details.

10.3.1 Error taxonomy for login flows

Classify failures into categories. A stable taxonomy makes logs searchable and helps operations.

Category A: User-caused (expected)

These are not “errors” in the operational sense; they are part of normal behavior:

- missing identifier or password,
- invalid identifier shape,
- wrong password,
- expired password (if policy exists),
- MFA required but not provided (if MFA is enabled).

Handling principle: show a generic message and allow immediate retry. Log at low severity and increment security counters.

Category B: Security-policy outcomes (controlled)

These represent deliberate policy enforcement:

- account locked due to failed attempts,
- account disabled/suspended,
- rate limit triggered (client or identifier-based),
- step-up required for a sensitive context.

Handling principle: do not reveal exact reason in a way that enables enumeration. Consider showing guidance that is *policy-safe* (e.g., “Try again later or contact support”) while recording full detail in audit logs.

Category C: System failures (unexpected)

These are operational issues:

- database unavailable or timeout,
- transaction failure,
- configuration missing (secrets, DB DSN),
- cryptographic library errors,
- internal exceptions and programming errors.

Handling principle: show a neutral error message, keep UI stable, avoid repeated expensive retries, and log at error level with correlation context.

10.3.2 Security rules for login error handling

The login page must obey these rules:

No account enumeration

Never let the response differ in a way that reveals whether the account exists:

- same UI message for “unknown user” and “wrong password”,
- similar response timing where practical,
- avoid distinct HTTP status codes when deployed behind proxies that expose them.

No sensitive internal details to the client

Never show:

- exception messages,
- database error strings,
- stack traces,
- configuration values,
- exact lockout thresholds or counters.

Stable and minimal feedback

Use a short set of messages with controlled meaning:

- **Validation failure**: “Please enter a valid identifier and password.”
- **Authentication failure**: “Login failed. Check your credentials and try again.”
- **Temporary restriction**: “Unable to sign in right now. Please try again later.”
- **System failure**: “Service is temporarily unavailable. Please try again later.”

10.3.3 View-level error handling pattern

The login view must:

- guard the handler with **try/catch**,
- restore UI controls (re-enable login button),
- clear or keep fields according to policy,

- show a safe feedback message.

A robust pattern is to make the login handler a **small orchestration function**:

- validate inputs,
- call the authentication service,
- map service outcomes to UI results,
- never let exceptions escape.

A typed outcome for UI mapping

```
#pragma once
#include <cstdint>
#include <string>

enum class LoginUiOutcome {
  Success,
  InvalidInput,
  AuthFailed,
  TemporarilyBlocked,
  SystemError
};

struct LoginUiResult {
  LoginUiOutcome outcome{LoginUiOutcome::SystemError};
  std::int64_t userId{0}; // valid only on Success
};
```

View handler using a safe mapping layer

```
#include <Wt/WApplication.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
#include <exception>

static void setFeedback(Wt::WText* out, const std::string& msg, bool isError) {
  out->setText(Wt::WString::fromUTF8(msg));
  out->setStyleClass(isError ? "auth-feedback-text error" :
  ↪  "auth-feedback-text ok");
}

template <typename Fn>
static void guardLoginHandler(Wt::WPushButton* btn, Wt::WText* feedback, Fn&& fn) {
  btn->setEnabled(false);
  try {
    fn();
  } catch (...) {
    setFeedback(feedback,
    ↪  "Service is temporarily unavailable. Please try again later.", true);
  }
  btn->setEnabled(true);
}
```

10.3.4 Service-level error handling and normalization

The authentication service must **normalize** its outcomes so the view can handle them without guessing. This means:

- the service should rarely throw for expected outcomes,
- policy outcomes (locked, rate limited) are returned as explicit codes,
- only truly unexpected failures throw (or return an error code that the view treats as system failure).

Service result codes

```
#pragma once
#include <cstdint>

enum class AuthCode {
  Ok,
  InvalidInput,
  InvalidCredentials,
  RateLimited,
  Locked,
  Disabled,
  BackendFailure
};

struct AuthResult {
  AuthCode code{AuthCode::BackendFailure};
  std::int64_t userId{0}; // valid only when code == Ok
};
```

Mapping service results to UI outcomes

```
static LoginUiResult mapAuthToUi(const AuthResult& r) {
  switch (r.code) {
```

```
    case AuthCode::Ok:
      return {LoginUiOutcome::Success, r.userId};
    case AuthCode::InvalidInput:
      return {LoginUiOutcome::InvalidInput, 0};
    case AuthCode::InvalidCredentials:
      return {LoginUiOutcome::AuthFailed, 0};
    case AuthCode::RateLimited:
    case AuthCode::Locked:
      return {LoginUiOutcome::TemporarilyBlocked, 0};
    case AuthCode::Disabled:
      // Do not leak ``disabled'' to anonymous user; treat as generic auth failure
      ↪  or blocked.
      return {LoginUiOutcome::AuthFailed, 0};
    case AuthCode::BackendFailure:
    default:
      return {LoginUiOutcome::SystemError, 0};
  }
}
```

10.3.5 Login view: complete error-handling flow

This example shows a complete **error-hardened** Wt login attempt:

- validates inputs,
- calls the auth service,
- maps to a UI-safe message,
- updates session state only on success,
- redirects only on success,

- never throws to the framework.

```
#include <string>

class AuthSessionState; // assumed from previous sections
class IAuthService2 {
public:
  virtual ~IAuthService2() = default;
  virtual AuthResult authenticate(const std::string& identifier,
                                  const std::string& password) = 0;
};

static void handleLoginAttempt(Wt::WLineEdit* identifier,
                               Wt::WLineEdit* password,
                               Wt::WPushButton* loginBtn,
                               Wt::WText* feedback,
                               IAuthService2& auth,
                               AuthSessionState& sessionState,
                               const std::string& postLoginPath)
{
  guardLoginHandler(loginBtn, feedback, [&] {
    // UI-level shape validation.
    const bool idOk = (identifier->validate() == Wt::ValidationState::Valid);
    const bool pwOk = (password->validate() == Wt::ValidationState::Valid);

    if (!idOk || !pwOk) {
      setFeedback(feedback, "Please enter a valid identifier and password.", true);
      return;
    }
```

```
  const std::string id = identifier->text().toUTF8();
  const std::string pw = password->text().toUTF8();

  // Service-level outcome normalization.
  AuthResult ar{};
  try {
    ar = auth.authenticate(id, pw);
  } catch (...) {
    // Treat unexpected exceptions as system error, do not leak details.
    setFeedback(feedback,
    ↪  "Service is temporarily unavailable. Please try again later.", true);
    return;
  }

  const LoginUiResult ui = mapAuthToUi(ar);

  switch (ui.outcome) {
    case LoginUiOutcome::Success: {
      // Update session state (permissions cache omitted here for brevity).
      sessionState.onLogin(ui.userId, PermissionCache{});
      setFeedback(feedback, "Signed in successfully.", false);

      // Redirect to protected page.
      Wt::WApplication::instance()->setInternalPath(postLoginPath, true);
      return;
    }
    case LoginUiOutcome::InvalidInput:
      setFeedback(feedback, "Please enter a valid identifier and password.",
      ↪  true);
```

```
        return;

      case LoginUiOutcome::AuthFailed:
        setFeedback(feedback,
        ↪  "Login failed. Check your credentials and try again.", true);
        return;

      case LoginUiOutcome::TemporarilyBlocked:
        setFeedback(feedback,
        ↪  "Unable to sign in right now. Please try again later.", true);
        return;

      case LoginUiOutcome::SystemError:
      default:
        setFeedback(feedback,
        ↪  "Service is temporarily unavailable. Please try again later.", true);
        return;
    }
  });
}
```

10.3.6 Logging and auditing errors safely

Error handling is incomplete without **observability**. However, logging itself can become a vulnerability if you log secrets.

Never log secrets

Do not log:

- passwords,

- raw reset tokens,
- full authentication headers,
- session cookies,
- private keys or signing material.

Log with correlation context

For operational triage, log:

- timestamp and severity,
- request correlation id (generated per request/session),
- outcome code (invalid credentials, rate-limited, backend failure),
- safe identifier representation (e.g., hashed or partially redacted),
- source context (IP/UA) only if your privacy policy and compliance allow it.

Audit security-relevant outcomes

Audit events for:

- login success,
- login failure (invalid credentials),
- rate limiting triggered,
- account lockout,
- administrative unlock/disable.

Audit is distinct from logs: it is structured, retained, and reviewable for security investigations.

10.3.7 Fail-safe behavior under backend outages

If the database is down:

- the login view must not spin in retries,
- the service should fail fast with `BackendFailure`,
- the UI should show a neutral message,
- the system should alert operations via metrics/logs.

If email/notification is down:

- login may still operate,
- password reset or verification flows should degrade gracefully (inform the user without exposing internals).

10.3.8 DoS-resistance considerations in error handling

Bad error handling can amplify attacks:

- expensive exception stack building on every request,
- heavy string formatting or template rendering in failure loops,
- database lookups for obviously invalid input,
- returning distinct behaviors that help attackers optimize attempts.

Mitigations:

- validate shape before database operations,
- normalize outcomes (few code paths),

- keep messages constant and simple,
- integrate rate limiting early.

10.3.9 Summary

A production Wt login page must treat errors as part of normal operation and handle them with:

- a clear taxonomy (expected vs policy vs system failures),
- strict anti-enumeration behavior (generic messages, stable outcomes),
- view-level guards that never leak exceptions to the client,
- service-level normalization (explicit result codes, minimal throwing),
- safe logging and auditing (no secrets, rich correlation),
- fail-safe behavior during outages and DoS attempts.

This design keeps the login experience stable for users while ensuring that the system remains secure, diagnosable, and resilient under real-world operational conditions.

10.4 Connecting the UI to the controller

In a production Wt application, the login page must be structured so that:

- UI widgets remain focused on presentation and user interaction,
- business logic (authentication, rate limiting, auditing, permissions) stays outside the view,
- errors are normalized into safe outcomes,

- the UI is driven by a small set of state transitions.

This section connects the **Login View** (widgets, validators, feedback) to a **controller** (or presenter/use-case handler) that orchestrates login logic. The controller is the boundary where you:

- validate and normalize inputs (defense in depth),
- call services (auth, permissions, audit),
- return explicit outcomes for UI rendering,
- manage navigation decisions (`/login` $\rightarrow$ intended target).

10.4.1 Why a controller in a Wt project

Wt is server-driven: widget events call C++ slots directly. Without structure, it is easy to end up with:

- authentication logic mixed into widgets,
- duplicated checks across pages,
- inconsistent error behavior,
- hard-to-test UI-driven logic.

A controller provides:

- **testability**: controller methods can be unit-tested without Wt widgets,
- **consistency**: one policy for validation, rate limiting, outcomes,
- **separation**: UI is a thin layer mapping user input to controller calls.

10.4.2 Controller responsibilities for the login flow

The login controller (use-case handler) is responsible for:

- Accept raw input values from the UI.
- Apply input normalization (trim, canonicalization rules).
- Perform a security-safe validation pass (shape checks).
- Call **authentication service**.
- On success:
  - create/update the session-auth state,
  - load effective permissions (if not already returned),
  - record audit event,
  - compute the post-login navigation target.
- On failure:
  - return a generic UI-safe outcome,
  - record audit event (with details internally),
  - optionally apply backoff hints (without leaking internals).

10.4.3 Contracts: request, response, and outcomes

Define a small request/response pair that the UI can call and interpret.

Login request

```
#pragma once
#include <string>

struct LoginRequest {
  std::string identifier;
  std::string password;
  bool rememberMe{false};        // user preference, policy applied server-side
  std::string nextPath;          // optional internal redirect target
};
```

Login outcome and response

Keep outcomes explicit and few.

```
#pragma once
#include <cstdint>
#include <string>

enum class LoginOutcome {
  Success,
  InvalidInput,
  Failed,              // invalid credentials or non-disclosable states
  TemporarilyBlocked,  // rate limited / lockout
  SystemError
};

struct LoginResponse {
  LoginOutcome outcome{LoginOutcome::SystemError};
  std::int64_t userId{0};        // valid only on Success
```

```
  std::string redirectPath;      // valid only on Success
  std::string uiMessage;         // safe for display
};
```

10.4.4 Service interfaces used by the controller

The controller composes services. Keep these as interfaces for testability.

Authentication service (normalized result)

```
#pragma once
#include <cstdint>
#include <string>

enum class AuthCode {
  Ok,
  InvalidInput,
  InvalidCredentials,
  RateLimited,
  Locked,
  Disabled,
  BackendFailure
};

struct AuthResult {
  AuthCode code{AuthCode::BackendFailure};
  std::int64_t userId{0};
};

class IAuthService {
```

```
public:
  virtual ~IAuthService() = default;
  virtual AuthResult authenticate(const std::string& identifier,
                                  const std::string& password) = 0;
};
```

Permission loader

```
#pragma once
#include <cstdint>
#include <string>
#include <unordered_set>

class PermissionCache {
public:
  void clear() { perms_.clear(); }
  void add(const std::string& p) { perms_.insert(p); }
  bool has(const std::string& p) const { return perms_.count(p) != 0; }
private:
  std::unordered_set<std::string> perms_;
};

class IPermissionService {
public:
  virtual ~IPermissionService() = default;
  virtual PermissionCache loadEffectivePermissions(std::int64_t userId) = 0;
};
```

Audit service (security events)

```
#pragma once
#include <cstdint>
#include <string>

enum class AuditEventType {
  LoginSuccess,
  LoginFailure,
  LoginRateLimited,
  LoginLocked
};

struct AuditEvent {
  AuditEventType type{AuditEventType::LoginFailure};
  std::int64_t actorUserId{0};       // 0 for anonymous attempts
  std::string identifierHint;        // safe representation (redacted/hashed)
  bool success{false};
};

class IAuditService {
public:
  virtual ~IAuditService() = default;
  virtual void record(const AuditEvent& e) = 0;
};
```

Session auth state

```
#pragma once
#include <cstdint>
```

```
#include <optional>

class AuthSessionState {
public:
  void onLogin(std::int64_t userId, PermissionCache perms) {
    userId_ = userId;
    perms_ = std::move(perms);
  }
  void onLogout() {
    userId_.reset();
    perms_.clear();
  }
  bool loggedIn() const { return userId_.has_value(); }
  std::int64_t userId() const { return *userId_; }
  const PermissionCache& permissions() const { return perms_; }

private:
  std::optional<std::int64_t> userId_;
  PermissionCache perms_;
};
```

10.4.5 Controller implementation

The controller should be pure C++ logic, independent from Wt widget types. This keeps it testable and reusable.

Normalization and safe checks

```
#pragma once
#include <string>
```

```
static inline std::string trimAsciiWhitespace(std::string s) {
  auto isWs = [](unsigned char c) {
    return c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\f' || c == '\v';
  };
  while (!s.empty() && isWs(static_cast<unsigned char>(s.front())))
  ↪  s.erase(s.begin());
  while (!s.empty() && isWs(static_cast<unsigned char>(s.back())))  s.pop_back();
  return s;
}

static inline bool containsControlChar(const std::string& s) {
  for (unsigned char c : s) {
    if (c < 0x20 || c == 0x7F) return true;
  }
  return false;
}

static inline bool isSafeInternalPath(const std::string& p) {
  if (p.empty() || p.size() > 256) return false;
  if (p[0] != '/') return false;
  if (p.find("://") != std::string::npos) return false;
  if (p.find('\\') != std::string::npos) return false;
  if (p.find('\n') != std::string::npos || p.find('\r') != std::string::npos)
  ↪  return false;
  return true;
}

static inline std::string redactIdentifier(const std::string& id) {
```

```
  // Minimal, safe redaction for logs/audit. Do not log raw identifiers unless
  ↪  policy allows.
  if (id.size() <= 3) return "***";
  return id.substr(0, 1) + "***" + id.substr(id.size() - 1);
}
```

LoginController

```
#pragma once
#include <string>
#include <utility>

class LoginController {
public:
  LoginController(IAuthService& auth,
                  IPermissionService& perms,
                  IAuditService& audit,
                  AuthSessionState& sessionState,
                  std::string defaultRedirect = "/dashboard")
    : auth_(auth),
      perms_(perms),
      audit_(audit),
      sessionState_(sessionState),
      defaultRedirect_(std::move(defaultRedirect)) {}

  LoginResponse login(LoginRequest req) {
    // Normalize inputs
    req.identifier = trimAsciiWhitespace(std::move(req.identifier));
    req.password   = std::move(req.password); // password should not be altered
    ↪  except size checks
```

```
  req.nextPath   = trimAsciiWhitespace(std::move(req.nextPath));

  // Defensive shape validation (do not rely on UI-only validation)
  if (!validIdentifierShape(req.identifier) || !validPasswordShape(req.password))
  ↪  {
    return make(LoginOutcome::InvalidInput,
                "Please enter a valid identifier and password.");
  }

  // Determine safe redirect target
  const std::string redirect = (isSafeInternalPath(req.nextPath) ? req.nextPath :
  ↪  defaultRedirect_);

  // Call authentication service with normalized inputs
  AuthResult ar{};
  try {
    ar = auth_.authenticate(req.identifier, req.password);
  } catch (...) {
    // Unexpected exceptions are system errors; do not leak details.
    return make(LoginOutcome::SystemError,
                "Service is temporarily unavailable. Please try again later.");
  }

  // Map auth outcomes to UI-safe results and internal audit events
  switch (ar.code) {
    case AuthCode::Ok: {
      // Load effective permissions and update session state
      PermissionCache pc{};
      try {
```

```
        pc = perms_.loadEffectivePermissions(ar.userId);
      } catch (...) {
        // If permission loading fails, treat as system error; do not partially
        ↪ log in.
        return make(LoginOutcome::SystemError,
                    "Service is temporarily unavailable. Please try again later.");
      }

      sessionState_.onLogin(ar.userId, std::move(pc));

      audit_.record(AuditEvent{
        AuditEventType::LoginSuccess,
        ar.userId,
        redactIdentifier(req.identifier),
        true
      });

      LoginResponse r = make(LoginOutcome::Success, "Signed in successfully.");
      r.userId = ar.userId;
      r.redirectPath = redirect;
      return r;
    }

    case AuthCode::InvalidInput:
      audit_.record(AuditEvent{
        AuditEventType::LoginFailure,
        0,
        redactIdentifier(req.identifier),
        false
```

```
      });
      return make(LoginOutcome::InvalidInput,
                  "Please enter a valid identifier and password.");

    case AuthCode::InvalidCredentials:
    case AuthCode::Disabled:
      // Do not disclose whether account exists, is disabled, etc.
      audit_.record(AuditEvent{
        AuditEventType::LoginFailure,
        0,
        redactIdentifier(req.identifier),
        false
      });
      return make(LoginOutcome::Failed,
                  "Login failed. Check your credentials and try again.");

    case AuthCode::RateLimited:
      audit_.record(AuditEvent{
        AuditEventType::LoginRateLimited,
        0,
        redactIdentifier(req.identifier),
        false
      });
      return make(LoginOutcome::TemporarilyBlocked,
                  "Unable to sign in right now. Please try again later.");

    case AuthCode::Locked:
      audit_.record(AuditEvent{
        AuditEventType::LoginLocked,
```

```
          0,
          redactIdentifier(req.identifier),
          false
        });
        return make(LoginOutcome::TemporarilyBlocked,
                    "Unable to sign in right now. Please try again later.");

      case AuthCode::BackendFailure:
      default:
        return make(LoginOutcome::SystemError,
                    "Service is temporarily unavailable. Please try again later.");
    }
  }

private:
  IAuthService& auth_;
  IPermissionService& perms_;
  IAuditService& audit_;
  AuthSessionState& sessionState_;
  std::string defaultRedirect_;

  static bool validIdentifierShape(const std::string& id) {
    if (id.size() < 3 || id.size() > 254) return false;
    if (containsControlChar(id)) return false;
    return true;
  }

  static bool validPasswordShape(const std::string& pw) {
    if (pw.empty()) return false;
```

```
    if (pw.size() > 256) return false; // cap to prevent abuse
    if (containsControlChar(pw)) return false; // optional
    return true;
  }

  static LoginResponse make(LoginOutcome o, const std::string& msg) {
    LoginResponse r{};
    r.outcome = o;
    r.uiMessage = msg;
    return r;
  }
};
```

10.4.6 Connecting Wt UI events to the controller

Now the Wt login view is a thin adapter:

- read widget values,
- call `LoginController::login()`,
- render `uiMessage`,
- redirect on success.

UI adapter pattern

```
#include <Wt/WApplication.h>
#include <Wt/WLineEdit.h>
#include <Wt/WCheckBox.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
```

```
static void setFeedback(Wt::WText* out, const std::string& msg, bool isError) {
  out->setText(Wt::WString::fromUTF8(msg));
  out->setStyleClass(isError ? "auth-feedback-text error" :
  ↪  "auth-feedback-text ok");
}

static void connectLoginUiToController(Wt::WLineEdit* identifier,
                                       Wt::WLineEdit* password,
                                       Wt::WCheckBox* rememberMe,
                                       Wt::WPushButton* loginBtn,
                                       Wt::WText* feedback,
                                       LoginController& controller,
                                       const std::string& nextPathFromRouter)
{
  auto attempt = [=, &controller]() {
    loginBtn->setEnabled(false);

    LoginRequest req;
    req.identifier = identifier->text().toUTF8();
    req.password   = password->text().toUTF8();
    req.rememberMe = rememberMe ? rememberMe->isChecked() : false;
    req.nextPath   = nextPathFromRouter;

    LoginResponse r{};
    try {
      r = controller.login(std::move(req));
    } catch (...) {
      // Controller should not throw, but the UI must remain safe even if it does.
```

```
      setFeedback(feedback,
      ↪  "Service is temporarily unavailable. Please try again later.", true);
      loginBtn->setEnabled(true);
      return;
    }

    const bool ok = (r.outcome == LoginOutcome::Success);
    setFeedback(feedback, r.uiMessage, !ok);

    if (ok) {
      // Redirect to target computed by controller
      Wt::WApplication::instance()->setInternalPath(r.redirectPath, true);
    }

    loginBtn->setEnabled(true);
  };

  loginBtn->clicked().connect(attempt);
  identifier->enterPressed().connect(attempt);
  password->enterPressed().connect(attempt);
}
```

10.4.7 Router integration: passing the intended destination

The **router** decides which pages are protected and where to send unauthenticated users. To support deep links:

- store the requested internal path when redirecting to `/login`,
- pass it to the login view as `nextPathFromRouter`,

- validate it in the controller (`isSafeInternalPath`).

Illustrative router capture pattern

```
#include <string>

class NavigationState {
public:
  void setNextAfterLogin(std::string p) { next_ = std::move(p); }
  std::string nextAfterLogin() const { return next_; }
private:
  std::string next_{"/dashboard"};
};

// In your route() method:
// if (!auth.loggedIn()) { nav.setNextAfterLogin(internalPath());
↪  setInternalPath("/login", true); }
```

10.4.8 Controller testing (without Wt)

Because the controller does not depend on Wt types, you can test it using mock services.

Minimal fake services for a unit test

```
#include <cassert>

class FakeAuth : public IAuthService {
public:
  AuthResult authenticate(const std::string& identifier,
                          const std::string& password) override {
    if (identifier == "user@example.com" && password == "pw") {
```

```
      return {AuthCode::Ok, 42};
    }
    return {AuthCode::InvalidCredentials, 0};
  }
};

class FakePerms : public IPermissionService {
public:
  PermissionCache loadEffectivePermissions(std::int64_t userId) override {
    PermissionCache p;
    if (userId == 42) {
      p.add("user.read.self");
      p.add("session.login");
    }
    return p;
  }
};

class FakeAudit : public IAuditService {
public:
  void record(const AuditEvent& e) override { last = e; }
  AuditEvent last{};
};

int main() {
  FakeAuth auth;
  FakePerms perms;
  FakeAudit audit;
  AuthSessionState state;
```

```
  LoginController c(auth, perms, audit, state, "/dashboard");

  LoginRequest req;
  req.identifier = "user@example.com";
  req.password   = "pw";
  req.nextPath   = "/dashboard";

  LoginResponse r = c.login(req);
  assert(r.outcome == LoginOutcome::Success);
  assert(state.loggedIn());
  assert(state.userId() == 42);
  assert(r.redirectPath == "/dashboard");
  return 0;
}
```

10.4.9 Summary

Connecting the UI to the controller in a Wt login page is the discipline that prevents business logic from leaking into widgets. The recommended structure is:

- a thin Wt view that gathers input and renders feedback,
- a controller that normalizes inputs, calls services, returns explicit outcomes,
- service interfaces that implement authentication, permissions, and auditing,
- a router that supplies the intended target path for post-login navigation.

This separation yields a login flow that is secure (anti-enumeration, normalized errors), maintainable (single policy enforcement), and testable (controller unit tests without Wt).

Authentication and Session Management

11.1 Login and logout flow

This section defines and implements the **end-to-end login and logout flow** for the live User Management System (UMS) built with Wt. In a production server-side web application, authentication and session management must satisfy:

- **Correctness**: user identity is established only after successful credential verification.
- **Security**: prevent session fixation, account enumeration, and unsafe redirects; protect cookies.
- **Consistency**: centralized policy for login success/failure outcomes and audit events.
- **Resilience**: robust behavior under partial failures, timeouts, and repeated login attempts.
- **User experience**: predictable redirects, clear state transitions, and correct navigation behavior.

Wt offers an event-driven, stateful programming model where each browser session maps to a server-side application session. This enables strong server authority over UI state, but it also demands careful handling of session identity and logout.

11.1.1 Key concepts: identity, authentication state, and session

Identity vs authentication state

- **Identity** is the durable database concept: the `User` record (primary key, status, profile).
- **Authentication state** is the runtime fact: *this session is logged in as user X*.

The login flow transitions the application from:

$$\textit{Anonymous session} \rightarrow \textit{Authenticated session (UserId} = X\textit{)}$$

The logout flow transitions back:

$$\textit{Authenticated session} \rightarrow \textit{Anonymous session}$$

Session scope

In the live project we model session at two levels:

- **Wt session (application instance)**: server-side state associated with a browser session.
- **Application session record** (optional, but recommended): a database record that tracks logins across devices and supports forced logout and session revocation.

11.1.2 Flow requirements

Login flow requirements

- Validate input shape before hitting the database.
- Authenticate using the authentication service.
- On success:

  - establish the user identity in the session state,
  - load and cache effective permissions (RBAC),
  - record audit event,
  - redirect to a validated internal target path.
- On failure:
  - return a generic message (avoid enumeration),
  - apply throttling policy (rate limit/backoff),
  - record audit event with internal details.

Logout flow requirements

- Invalidate the authentication state.
- Invalidate/revoke any persistent login token (if supported).
- Optionally record logout in audit/session table.
- Redirect to a safe page (login or home).

Security requirements (non-negotiable)

- **Session fixation defense**: do not allow an attacker to force a victim to reuse a pre-authentication session identity.
- **Cookie protection**: Secure + HttpOnly + SameSite and correct path/domain.
- **Open redirect prevention**: redirect only to internal paths that pass a strict allow policy.
- **Least disclosure**: generic failure messages; do not reveal locked/disabled state to anonymous users.

- **Re-auth for sensitive actions**: require recent authentication for password changes, role changes, and administrative actions.

11.1.3 State machine: login/logout transitions

Define explicit session states:

- **Anonymous**: no `userId` and no permission cache.
- **Authenticated**: `userId` present and permission cache loaded.
- **Expired/Revoked**: session exists but is no longer valid (forced logout, rotation).

A minimal internal state object:

```
#pragma once
#include <cstdint>
#include <optional>
#include <string>
#include <unordered_set>

class PermissionCache {
public:
  void clear() { perms_.clear(); }
  void add(const std::string& p) { perms_.insert(p); }
  bool has(const std::string& p) const { return perms_.count(p) != 0; }
private:
  std::unordered_set<std::string> perms_;
};

class AuthSessionState {
```

```
public:
  void onLogin(std::int64_t userId, PermissionCache perms) {
    userId_ = userId;
    perms_ = std::move(perms);
  }

  void onLogout() {
    userId_.reset();
    perms_.clear();
  }

  bool loggedIn() const { return userId_.has_value(); }
  std::int64_t userId() const { return *userId_; }
  const PermissionCache& permissions() const { return perms_; }

private:
  std::optional<std::int64_t> userId_;
  PermissionCache perms_;
};
```

11.1.4 Capturing the intended destination (post-login redirect)

A robust system supports deep links to protected pages:

- User requests `/admin/users`.
- Router sees unauthenticated user and redirects to `/login`.
- After login, user is redirected back to `/admin/users`.

This requires storing a `nextAfterLogin` internal path in a server-side navigation state.

```
#pragma once
#include <string>

class NavigationState {
public:
  void setNextAfterLogin(std::string p) { next_ = std::move(p); }
  std::string nextAfterLogin() const { return next_; }
  void clearNextAfterLogin() { next_.clear(); }

private:
  std::string next_{"/dashboard"};
};

static bool isSafeInternalPath(const std::string& p) {
  if (p.empty() || p.size() > 256) return false;
  if (p[0] != '/') return false;
  if (p.find("://") != std::string::npos) return false;
  if (p.find('\\') != std::string::npos) return false;
  if (p.find('\n') != std::string::npos || p.find('\r') != std::string::npos)
  ↪  return false;
  return true;
}
```

11.1.5 Router enforcement of authentication

Every protected route must enforce authentication in the router (and again in actions). A clean pattern:

```
#include <Wt/WApplication.h>
#include <string>
```

```
static bool isProtectedPath(const std::string& path) {
  // Expand as your project grows.
  return path.rfind("/dashboard", 0) == 0 ||
         path.rfind("/admin", 0) == 0 ||
         path.rfind("/settings", 0) == 0;
}

static void enforceAuthOrRedirect(const std::string& requestedPath,
                                  AuthSessionState& auth,
                                  NavigationState& nav)
{
  if (isProtectedPath(requestedPath) && !auth.loggedIn()) {
    nav.setNextAfterLogin(requestedPath);
    Wt::WApplication::instance()->setInternalPath("/login", true);
  }
}
```

11.1.6 Login flow implementation

The login flow is orchestrated by the controller (from Chapter 10). This section focuses on the **session transition** and **navigation** aspects.

Controller result and post-login redirect

We assume a controller returns:

- a `LoginOutcome`,
- optional `userId`,

- and a `redirectPath` already validated as internal/safe.

On success:

- update `AuthSessionState`,
- clear `nextAfterLogin`,
- redirect to target.

```
#include <Wt/WApplication.h>

static void onLoginSuccess(std::int64_t userId,
                           AuthSessionState& auth,
                           NavigationState& nav,
                           PermissionCache perms,
                           const std::string& fallback = "/dashboard")
{
  auth.onLogin(userId, std::move(perms));

  std::string target = nav.nextAfterLogin();
  nav.clearNextAfterLogin();

  if (!isSafeInternalPath(target)) {
    target = fallback;
  }

  // Redirect to the intended page.
  Wt::WApplication::instance()->setInternalPath(target, true);
}
```

Remember-me (policy hook)

Remember-me is not a UI feature; it is a policy feature:

- it changes session lifetime and revocation behavior,
- it may require issuing a persistent token tied to the device.

A safe design is to treat it as:

- **server policy decision**,
- requiring revocation support and rotation.

In the live project, you can implement remember-me as an optional persistent token stored server-side (database) and delivered via secure cookie. If you do not implement it, ignore the checkbox and keep a short session lifetime.

11.1.7 Logout flow implementation

Logout must reliably clear authentication state and prevent the session from silently remaining authorized.

Logout actions

- Clear `AuthSessionState`.
- Clear navigation state (optional).
- Revoke persistent tokens (if any).
- Redirect to `/login` or public home.

Logout handler in Wt

```
#include <Wt/WApplication.h>
#include <Wt/WPushButton.h>

class LogoutController {
public:
  explicit LogoutController(AuthSessionState& auth, NavigationState& nav)
    : auth_(auth), nav_(nav) {}

  void logout() {
    // If you support persistent tokens, revoke them here (server-side record).
    auth_.onLogout();
    nav_.clearNextAfterLogin();
    Wt::WApplication::instance()->setInternalPath("/login", true);
  }

private:
  AuthSessionState& auth_;
  NavigationState& nav_;
};
```

Wiring logout into the UI

A logout action is often placed in a top navigation bar. The UI must call the logout controller.

```
#include <Wt/WContainerWidget.h>
#include <Wt/WPushButton.h>

static void addLogoutButton(Wt::WContainerWidget* parent, LogoutController& c) {
  auto btn = parent->addWidget(std::make_unique<Wt::WPushButton>("Logout"));
```

```
  btn->clicked().connect([&c] { c.logout(); });
}
```

11.1.8 Session invalidation and forced logout

In production you often need to force logout users:

- after password reset,
- after role/permission changes,
- after suspected compromise,
- after administrative action.

This is easiest if you maintain a **server-side session version** or **session record**:

- store a `sessionVersion` integer on the user record,
- store the session's version at login,
- on each sensitive request (or periodically), verify session version matches the user record,
- increment `sessionVersion` to revoke all sessions.

Session version pattern

```
#include <cstdint>

struct SessionClaims {
  std::int64_t userId{0};
  std::int64_t sessionVersion{0}; // snapshot at login time
};
```

```
static bool isSessionStillValid(const SessionClaims& claims,
                                std::int64_t currentUserSessionVersionFromDb)
{
  return claims.sessionVersion == currentUserSessionVersionFromDb;
}
```

When invalid:

- clear auth state,
- redirect to login,
- optionally show a message: “Your session has expired. Please sign in again.”

11.1.9 Recommended flow checks

After login

- rotate session identifiers if supported by the connector/session policy,
- cache permissions, but invalidate on role/permission changes,
- clear sensitive input fields (password) from UI objects.

Before logout completes

- ensure server-side tokens are revoked (if persistent),
- record audit event (logout),
- avoid redirecting to user-controlled URLs.

11.1.10 Complete flow summary (sequence)

Login sequence

- User requests a protected internal path.
- Router redirects to `/login` and stores `nextAfterLogin`.
- User submits credentials.
- UI validates shape, calls controller.
- Controller:
  - normalizes input,
  - authenticates,
  - loads permissions,
  - updates session state,
  - returns a safe redirect path.
- UI redirects to the original protected page.

Logout sequence

- User triggers logout (button/menu).
- Logout controller clears auth state and revokes persistent tokens.
- Redirect to `/login`.
- Any subsequent access to protected paths triggers router enforcement.

11.1.11 Summary

A correct login/logout flow in a Wt-based user system is built from:

- **router enforcement** (protected internal paths redirect to login),
- **controller-driven login** (normalized outcomes, permissions cache, audit),
- **explicit session state** (authenticated vs anonymous),
- **safe redirects** (internal path validation),
- **reliable logout** (state clearing and token revocation),
- **optional forced logout mechanisms** (session version or session records).

This structure yields a secure, testable authentication flow that behaves predictably under real production conditions.

11.2 Session state handling

Session state handling is a core foundation of a production-grade authentication system. In Wt, a session represents the server-side object lifecycle tied to a browser connection. Correct session state management ensures that authenticated users retain identity and permissions throughout their interaction, while also protecting against session fixation, stale credentials, and unauthorized access.

This section presents the design, implementation, and operational considerations to handle session state reliably, securely, and in a way that supports multi-device logins, permission changes, and forced logout.

11.2.1 Session state goals

A robust session state strategy must satisfy:

- **Correct representation of identity**: the server must track exactly who is authenticated and with what scope.
- **Permission caching and invalidation**: avoid repeated database calls while respecting changes in authorization policies.
- **Session revocation and expiration**: support logout, forced logout, and expiration policies.
- **Consistency across requests**: identity and permissions must be immutable for the duration of a single request.
- **Security hygiene**: mitigation against fixation, replay, and elevation attacks.
- **Testability**: the session abstraction should be unit-testable, independent of transport connectors.

11.2.2 Wt session lifecycle

In Wt, each browser connection is backed by a server-side application session (unless you configure sticky sessions). That object encapsulates:

- UI state and widgets,
- secure cookie state (session ID),
- application-specific session information (authentication, user context),
- event signal connections.

The application must explicitly manage authentication state within this session scope.

11.2.3 Session identity structure

Define a minimal structure for representing logged-in identity and permissions:

```
#pragma once
#include <cstdint>
#include <optional>
#include <string>
#include <unordered_set>

class PermissionCache {
public:
  void clear() { perms_.clear(); }
  void add(const std::string& p) { perms_.insert(p); }
  bool has(const std::string& p) const { return perms_.count(p) != 0; }
private:
  std::unordered_set<std::string> perms_;
};

class AuthSessionState {
public:
  void onLogin(std::int64_t userId, PermissionCache perms) {
    userId_ = userId;
    perms_ = std::move(perms);
    sessionVersion_ = currentTimeMillis();
  }

  void onLogout() {
    userId_.reset();
    perms_.clear();
```

```
    sessionVersion_ = 0;
  }

  bool loggedIn() const { return userId_.has_value(); }
  std::int64_t userId() const { return *userId_; }
  const PermissionCache& permissions() const { return perms_; }

  std::int64_t sessionVersion() const { return sessionVersion_; }

private:
  std::optional<std::int64_t> userId_;
  PermissionCache perms_;
  std::int64_t sessionVersion_{0};

  static std::int64_t currentTimeMillis() {
    using namespace std::chrono;
    return
    ↪  duration_cast<milliseconds>(system_clock::now().time_since_epoch()).count();
  }
};
```

11.2.4 Session versioning and forced logout

A key requirement in many production systems is the ability to force logout a user across devices or after a security event (password change, role reassignment, compromise detection). To support forced logout, maintain a session version:

- store a `sessionVersion` in the user record in the database,
- include it in the session state at login time,

- check it on each request or at key boundaries,
- increment the database's `sessionVersion` to invalidate all existing sessions.

This pattern decouples session tokens from permanent validity and gives administrators control over session revocation.

```
#include <cstdint>

static bool isSessionVersionValid(std::int64_t sessionVersionStored,
                                  std::int64_t sessionVersionInSession) {
  return sessionVersionStored == sessionVersionInSession;
}
```

If invalid:

- clear `AuthSessionState`,
- redirect to login with an optional message,
- record audit event (session invalidation).

11.2.5 Session timeout policies

Define two standard timeouts:

- **Idle timeout**: how long a session can remain idle before requiring re-authentication.
- **Absolute timeout**: maximum lifetime for a session regardless of activity.

In the session state object:

```
#include <chrono>

class SessionTimeoutPolicy {
public:
  SessionTimeoutPolicy(std::chrono::minutes idle,
                       std::chrono::hours absolute)
    : idleTimeout_{idle}, absoluteTimeout_{absolute} {}

  bool isExpired(std::int64_t lastAccessMillis,
                 std::int64_t loginMillis) const {
    const auto now = currentTimeMillis();
    const auto idleElapsed = now - lastAccessMillis;
    const auto absoluteElapsed = now - loginMillis;
    return (idleElapsed > toMillis(idleTimeout_)) ||
           (absoluteElapsed > toMillis(absoluteTimeout_));
  }

private:
  std::chrono::minutes idleTimeout_;
  std::chrono::hours absoluteTimeout_;

  static std::int64_t toMillis(std::chrono::milliseconds d) {
    return d.count();
  }

  static std::int64_t currentTimeMillis() {
    using namespace std::chrono;
    return
    ↪  duration_cast<milliseconds>(system_clock::now().time_since_epoch()).count();
```

```
  }
};
```

Track `lastAccessMillis` and `loginMillis` in session state to enforce timeouts. Update `lastAccessMillis` on each significant request.

11.2.6 Cookie-based session identifier

Unless a persistent token is implemented, Wt uses a secure cookie to track the session identifier. Ensure:

- **Secure** attribute is set (HTTPS only),
- **HttpOnly** is set (not accessible to JS),
- **SameSite** is set to `Lax` or `Strict` (to mitigate CSRF),
- cookie path/domain align with application scope.

Session identifiers must be unpredictable and unique. Do not derive them from user identity or weak entropy.

11.2.7 Session storage considerations

Wt stores session state in memory by default. In distributed environments (multi-instance behind a load balancer), consider:

- **sticky sessions**: traffic routed to the same instance,
- **session replication**: shared backend for session state (Redis, database),
- **stateless tokens**: JSON Web Tokens (JWTs) or similar, with server-side validation and revocation support.

For UMS, a recommended approach is:

- primary in-memory session for full state (widgets, UI),
- persistent session records in the database for authentication ticketing and revocation.

A session record may contain:

```
#pragma once
#include <cstdint>
#include <string>

struct SessionRecord {
  std::string sessionId;           // unique identifier
  std::int64_t userId;             // associated user
  std::int64_t loginTimeMillis;    // login moment
  std::int64_t lastAccessMillis;   // updated on activity
  std::int64_t sessionVersion;     // snapshot of user session version
};
```

Use database indexes on `sessionId` and `userId` for efficient lookup and revocation.

11.2.8 Initialization and teardown

At login

- generate a new session identifier (or rotate existing),
- update `AuthSessionState` with identity and permission cache,
- create a persistent session record with timestamps and version snapshot,
- set secure session cookie.

At logout

- remove the persistent session record,
- clear `AuthSessionState`,
- expire session cookie.

11.2.9 Periodic validation hooks

To enforce revocation and timeouts consistently, insert session validation checks at key request boundaries:

- before rendering a protected route,
- before performing a sensitive action.

```
static bool validateSession(AuthSessionState& auth,
                            const SessionRecord& record,
                            const SessionTimeoutPolicy& policy)
{
  if (!auth.loggedIn()) return false;
  if (!isSessionVersionValid(record.sessionVersion, auth.sessionVersion())) {
    return false;
  }
  if (policy.isExpired(record.lastAccessMillis, record.loginTimeMillis)) {
    return false;
  }
  return true;
}
```

If invalid:

- clear state,
- redirect to login,
- optionally add a feedback message.

11.2.10 Session-based permission caching

Loading permissions on every request is expensive. Cache them in session state:

- on login,
- after explicit role/permission changes (invalidate cache),
- on forced logout or session expiry.

Invalidate the cache if:

- the server detects a role/permission change for the user,
- the user performs a sensitive operation that should re-evaluate policy.

11.2.11 Sensitive operations and re-authentication

For high-risk actions (password change, role assignments), enforce re-authentication:

- check the session age or last authentication timestamp,
- prompt for credentials again if needed,
- update session state on success.

11.2.12 Session cleanup and garbage collection

In-memory sessions and persistent session records accumulate. Implement cleanup policies:

- expire idle sessions after a threshold,
- purge absolute expired sessions,
- periodic database cleanup job.

Cleanups improve performance and reduce security risks from stale sessions.

11.2.13 Testing session state logic

Unit-test session logic independently of Wt widgets. Example:

```
#include <cassert>

int main() {
  AuthSessionState s;
  assert(!s.loggedIn());

  PermissionCache pc;
  pc.add("test");
  s.onLogin(42, pc);
  assert(s.loggedIn());
  assert(s.userId() == 42);
  assert(s.permissions().has("test"));

  s.onLogout();
  assert(!s.loggedIn());
```

```
  return 0;
}
```

11.2.14 Summary

Session state handling in a Wt-based user system encompasses:

- managing identity and permissions in a secure session object,
- session versioning for forced logout,
- timeout policies (idle and absolute),
- persistent session records for multi-instance environments,
- secure cookie management,
- validation hooks for revocation and timeouts,
- periodic cleanup and resource management,
- clear separation between in-memory session and persistent records.

A disciplined approach to session state handling yields a system that is secure, scalable, testable, and maintainable in production environments.

11.3 Protecting secured pages

Protecting secured pages is the practical enforcement layer of authentication and authorization. A user management system is only correct if **every protected capability** is enforced on the server, regardless of what the browser renders. In Wt, the application is server-side and event-driven, but you still must assume:

- users can craft requests and internal paths manually,
- UI hiding alone is not security,
- sessions may become stale (role changes, forced logout, timeouts),
- sensitive state must be re-validated at every privileged boundary.

This section defines a production-grade approach for protecting secured pages in a Wt application using:

- centralized route protection (internal path router),
- consistent authorization checks in controllers/services,
- safe redirection and no open redirects,
- session validity checks (timeouts, revocation, account state),
- UI-level gating as a secondary usability layer.

11.3.1 Threat model for secured pages

A minimal threat model for protected pages includes:

- **Direct navigation**: user types `/admin/users` in the URL.
- **Stale privilege**: user was admin, role revoked, but session still cached.
- **Session theft**: attacker obtains a session cookie (mitigated by cookie attributes and TLS).
- **Session fixation**: attacker tries to force the victim to reuse a known session identifier.
- **Open redirect**: attacker supplies a malicious redirect target during login.

- **CSRF on state changes**: attacker triggers privileged actions if cookies are sent automatically.

Your design must ensure that:

- protected pages never render for unauthenticated sessions,
- privileged actions never execute without permission checks,
- session validity is continuously enforced.

11.3.2 Protection layers (defense in depth)

Use multiple layers that each remain correct on their own:

Layer 1: Route protection (navigation boundary)

- block access to protected internal paths when not logged in,
- redirect to login and remember the originally requested path,
- validate the redirect target strictly.

Layer 2: Page construction protection (render boundary)

Even if a user reaches a page instance, the page must:

- verify authentication and account state again,
- render a forbidden/redirect view safely.

Layer 3: Action protection (execution boundary)

Every privileged action (buttons, REST endpoints, Wt events) must:

- require permissions server-side,
- perform record-level checks (ownership/scope),
- validate session freshness for sensitive operations (step-up if required).

11.3.3 Central route guard using internal paths

Wt uses `internalPath` for routing. A central router should:

- know which paths are protected,
- enforce authentication before rendering them,
- send anonymous sessions to `/login`.

Protected path policy

Start with a clear policy function. Avoid scattered string checks.

```
#include <string>

static bool isProtectedPath(const std::string& path) {
  // Prefix-based protection is simple and effective for internal routing.
  // Expand with your real routes.
  return path.rfind("/dashboard", 0) == 0 ||
         path.rfind("/settings", 0) == 0 ||
         path.rfind("/admin", 0) == 0;
}
```

Safe redirect target validation

Never redirect to user-supplied external URLs. Only internal paths are allowed.

```
#include <string>

static bool isSafeInternalPath(const std::string& p) {
  if (p.empty() || p.size() > 256) return false;
  if (p[0] != '/') return false;
  if (p.find("://") != std::string::npos) return false;
  if (p.find('\\') != std::string::npos) return false;
  if (p.find('\n') != std::string::npos || p.find('\r') != std::string::npos)
  ↪  return false;
  return true;
}
```

Navigation state: remember original destination

```
#include <string>

class NavigationState {
public:
  void setNextAfterLogin(std::string p) { next_ = std::move(p); }
  std::string nextAfterLogin() const { return next_; }
  void clearNextAfterLogin() { next_.clear(); }

private:
  std::string next_{"/dashboard"};
};
```

Auth guard invoked by router

```
#include <Wt/WApplication.h>

class AuthSessionState {
public:
  bool loggedIn() const { return loggedIn_; }
  void onLogin() { loggedIn_ = true; }
  void onLogout() { loggedIn_ = false; }
private:
  bool loggedIn_{false};
};

static void enforceAuthOrRedirect(const std::string& requestedPath,
                                  AuthSessionState& auth,
                                  NavigationState& nav)
{
  if (isProtectedPath(requestedPath) && !auth.loggedIn()) {
    // Store only validated internal path; fall back to a safe default otherwise.
    nav.setNextAfterLogin(isSafeInternalPath(requestedPath) ? requestedPath :
    ↪  "/dashboard");
    Wt::WApplication::instance()->setInternalPath("/login", true);
  }
}
```

11.3.4 Continuous session validity checks

Authenticated sessions can become invalid due to:

- idle/absolute timeouts,

- role changes or forced logout,
- account lock/disable,
- password resets that revoke sessions.

Therefore, before showing protected pages, validate session state:

Session validity contract

```
#include <cstdint>

struct SessionClaims {
  std::int64_t userId{0};
  std::int64_t sessionVersion{0}; // snapshot at login time
  std::int64_t loginMillis{0};
  std::int64_t lastAccessMillis{0};
};

struct UserSecuritySnapshot {
  bool active{true};
  bool locked{false};
  std::int64_t currentSessionVersion{0};
};

static bool isSessionStillValid(const SessionClaims& c,
                                const UserSecuritySnapshot& u,
                                std::int64_t nowMillis,
                                std::int64_t idleTimeoutMillis,
                                std::int64_t absoluteTimeoutMillis)
{
```

```
  if (c.userId == 0) return false;
  if (!u.active || u.locked) return false;
  if (c.sessionVersion != u.currentSessionVersion) return false;

  const auto idleAge = nowMillis - c.lastAccessMillis;
  const auto absAge  = nowMillis - c.loginMillis;
  if (idleAge > idleTimeoutMillis) return false;
  if (absAge > absoluteTimeoutMillis) return false;
  return true;
}
```

Invalid session handling

If invalid:

- clear authentication state,
- redirect to `/login`,
- optionally show a generic message (`session expired`).

```
#include <Wt/WApplication.h>

static void invalidateAndRedirect(AuthSessionState& auth, NavigationState& nav) {
  auth.onLogout();
  nav.clearNextAfterLogin();
  Wt::WApplication::instance()->setInternalPath("/login", true);
}
```

11.3.5 Authorization checks: permissions and scope

Authentication answers: *who is the user?* Authorization answers: *what is the user allowed to do?*

Protected pages typically require both:

- the user is logged in,
- the user has a required permission (RBAC),
- if applicable, the user is within the correct scope/tenant,
- for sensitive pages, the session is recent enough (step-up).

Permission gate helper

```
#include <stdexcept>
#include <string>

class PermissionCache {
public:
  bool has(const std::string& p) const { return perms_.count(p) != 0; }
  void add(std::string p) { perms_.insert(std::move(p)); }
private:
  std::unordered_set<std::string> perms_;
};

static void requirePermission(const PermissionCache& perms, const std::string& p) {
  if (!perms.has(p)) {
    throw std::runtime_error("Forbidden");
  }
}
```

Page-level permission requirement

For example, the admin user list page might require `user.read.any`:

```
static void requireAdminUsersAccess(const AuthSessionState& auth,
                                    const PermissionCache& perms)
{
  if (!auth.loggedIn()) {
    throw std::runtime_error("Unauthenticated");
  }
  requirePermission(perms, "user.read.any");
}
```

11.3.6 Router + page factory pattern

A maintainable Wt app often uses a page factory that creates widgets based on internal path. This factory can enforce protection rules centrally.

Protected page creation

```
#include <Wt/WContainerWidget.h>
#include <Wt/WText.h>
#include <memory>
#include <string>

static std::unique_ptr<Wt::WContainerWidget> makeForbiddenView() {
  auto w = std::make_unique<Wt::WContainerWidget>();
  w->addWidget(std::make_unique<Wt::WText>("Access denied."));
  return w;
}
```

```
static std::unique_ptr<Wt::WContainerWidget> makeLoginRequiredView() {
  auto w = std::make_unique<Wt::WContainerWidget>();
  w->addWidget(std::make_unique<Wt::WText>("Please sign in to continue."));
  return w;
}

static std::unique_ptr<Wt::WContainerWidget> makeDashboard() {
  auto w = std::make_unique<Wt::WContainerWidget>();
  w->addWidget(std::make_unique<Wt::WText>("Dashboard"));
  return w;
}

static std::unique_ptr<Wt::WContainerWidget> buildPageForPath(const std::string&
↪  path,
                                                              AuthSessionState& auth,
                                                               const PermissionCache&
                                                                 ↪  perms)
{
  // Example routing:
  if (path == "/dashboard") {
    if (!auth.loggedIn()) return makeLoginRequiredView();
    return makeDashboard();
  }

  if (path == "/admin/users") {
    if (!auth.loggedIn()) return makeLoginRequiredView();
    if (!perms.has("user.read.any")) return makeForbiddenView();
    auto w = std::make_unique<Wt::WContainerWidget>();
    w->addWidget(std::make_unique<Wt::WText>("Admin Users Page"));
```

```
    return w;
  }

  // Default:
  return makeLoginRequiredView();
}
```

11.3.7 UI-level protection (secondary layer)

UI gating improves UX but is not the enforcement boundary.

Conditional widget visibility

```
#include <Wt/WWidget.h>

static void setVisibleIfAllowed(Wt::WWidget* w,
                                const PermissionCache& perms,
                                const std::string& perm)
{
  w->setVisible(perms.has(perm));
}
```

Use this for:

- showing an Admin menu only to admins,
- disabling buttons the user cannot use.

But still enforce permissions when the action runs.

11.3.8 Protecting actions: never trust the UI

Every privileged action must re-check:

- authentication state,
- permission,
- record-level scope.

Example: protected handler

```
#include <Wt/WPushButton.h>

static void connectDeleteUserAction(Wt::WPushButton* btn,
                                    AuthSessionState& auth,
                                    const PermissionCache& perms,
                                    std::function<void()> performDelete)
{
  btn->clicked().connect([&] {
    if (!auth.loggedIn()) {
      throw std::runtime_error("Unauthenticated");
    }
    requirePermission(perms, "user.disable");
    performDelete();
  });
}
```

In production, catch exceptions at boundaries and show safe messages rather than propagating errors.

11.3.9 CSRF and state-changing operations

Even with SameSite cookies, treat CSRF as a relevant concern for state changes. A robust design uses:

- CSRF tokens for form submissions that modify state,
- same-origin internal paths,
- verification on the server for every state-changing request.

In Wt, many interactions are event-based and go through the framework's request handling. Still, the application should implement CSRF tokens for critical actions if it exposes endpoints or forms beyond Wt event channels.

11.3.10 Production checklist for secured pages

- Central router identifies protected paths and redirects unauthenticated users.
- Redirect targets are validated internal paths (no open redirects).
- Session validity is checked periodically (timeouts, revocation, account state).
- Permissions are enforced at action execution, not only UI rendering.
- Permission cache is invalidated on role/permission changes (or session version revocation).
- Sensitive operations require step-up authentication.
- Forbidden and login-required views do not leak internal details.
- Errors are handled safely and logged/audited without secrets.

11.3.11 Summary

Protecting secured pages in a Wt-based UMS is a layered strategy:

- **route protection** to stop unauthorized navigation early,
- **page-level checks** to prevent accidental rendering of protected content,
- **action-level authorization** to enforce policy regardless of UI state,
- **session validity enforcement** to handle revocation and timeouts,
- **safe redirects and messages** to prevent open redirects and enumeration.

This approach yields predictable, testable security boundaries that remain correct as the project grows.

11.4 User context management

A production authentication system is not complete when you can merely log in and log out. After authentication, every secured operation needs a **consistent, safe, and testable view of the current user**. This is the purpose of **user context management**.
In the live User Management System (UMS), the **User Context** is the canonical runtime object that represents:

- **who** is the current user (identity claims),
- **what** the user can do (permissions and scope),
- **under which constraints** the user acts (account state, tenant, session validity),
- **how** the request should be treated (locale/timezone, correlation id, step-up freshness).

Correct user context management prevents common failures:

- scattered and inconsistent permission checks,
- stale authorization decisions (role changes not reflected),
- mixing user profile reads with security-critical claims,
- accidental trust in UI state instead of server-side policy.

11.4.1 Design goals

A robust user context system must satisfy:

- **Single source of truth**: controllers and pages use the same context object.
- **Minimal trusted claims**: store only what is required for authorization and routing decisions.
- **Immutable-per-request**: context should not change during a single request/event handler.
- **Explicit refresh**: when policy changes (roles/locks), refresh or invalidate deterministically.
- **Fast permission checks**: constant-time permission membership checks via cached set/bitset.
- **Separation of concerns**: keep profile/UI preferences separate from security decisions.
- **Testability**: context creation and authorization gates can be unit-tested without Wt widgets.

11.4.2 User context: what it contains

A practical context typically splits into:

Security-critical claims

- `userId` (durable identity key),
- `sessionVersion` snapshot (revocation/forced logout),
- account flags: `active`, `locked`,
- `authTime` and `lastAuthTime` (step-up requirements),
- `permissions` (effective permissions cache),
- scope identifiers (tenant/project/org) if multi-tenant.

Non-security preferences (still useful)

- display name,
- locale and timezone,
- UI preferences (theme), if needed.

Request correlation and diagnostics

- request id / correlation id (for logs and audit),
- optional device/session id (for session record tracking),
- optional client hints (sanitized).

11.4.3 C++ model: UserContext and PermissionCache

```
#pragma once
#include <cstdint>
#include <string>
#include <unordered_set>
#include <optional>

class PermissionCache {
public:
  void clear() { perms_.clear(); }
  void add(const std::string& p) { perms_.insert(p); }
  bool has(const std::string& p) const { return perms_.count(p) != 0; }

private:
  std::unordered_set<std::string> perms_;
};

struct UserScope {
  // Single-tenant systems can omit this.
  // For multi-tenant: tenantId is a security boundary.
  std::optional<std::int64_t> tenantId;
  std::optional<std::int64_t> orgId;
  std::optional<std::int64_t> projectId;
};

struct UserContext {
  // --- Security-critical identity claims ---
  std::int64_t userId{0};              // 0 => anonymous
```

```
  std::int64_t sessionVersion{0};      // snapshot at login time
  bool active{false};
  bool locked{false};

  // Step-up freshness (optional but recommended for sensitive operations)
  std::int64_t authTimeMillis{0};      // when user last proved identity
  ↪  (password/MFA)
  std::int64_t lastAccessMillis{0};    // activity timestamp for idle timeout policy

  PermissionCache perms;
  UserScope scope;

  // --- Non-security profile/preferences ---
  std::string displayName;
  std::string locale;                  // e.g., "en", "ar"
  std::string timeZone;                // e.g., "Asia/Riyadh"

  // --- Diagnostics / auditing ---
  std::string requestId;               // correlation id for logs/audit
  std::string sessionIdHint;           // safe identifier (not raw cookie)

  bool authenticated() const { return userId != 0; }
};
```

11.4.4 Where the context lives in a Wt application

In Wt, the natural place for session-wide state is your **application instance** (one per browser session). The context is typically stored in:

- a session state object (`AuthSessionState`),

- plus a derived, richer `UserContext` built from the database on login or on demand.

Important rules:

- Do not rebuild context repeatedly in UI code.
- Do not query the database inside every widget event handler if you can avoid it.
- Keep the session context minimal and refresh it deliberately.

11.4.5 Context provider pattern

Use a dedicated component to build, validate, cache, and refresh the context:

- **Context Provider** loads user security snapshot + permissions from the database.
- It enforces session validity rules (version match, lock/disable, timeouts).
- It returns a **ready-to-use** `UserContext` to controllers/pages.

Service contracts

```
#pragma once
#include <cstdint>
#include <string>

struct UserSecuritySnapshot {
  std::int64_t userId{0};
  bool active{false};
  bool locked{false};
  std::int64_t sessionVersion{0};
  std::string displayName;
  std::string locale;
```

```
  std::string timeZone;
};

class IUserRepository {
public:
  virtual ~IUserRepository() = default;
  virtual UserSecuritySnapshot loadSecuritySnapshot(std::int64_t userId) = 0;
  virtual UserScope loadScope(std::int64_t userId) = 0;
};

class IPermissionService {
public:
  virtual ~IPermissionService() = default;
  virtual PermissionCache loadEffectivePermissions(std::int64_t userId) = 0;
};
```

Session claims and validity policy

Session claims are what your session remembers *without* needing DB calls on every action. A common minimal set is:

- `userId`,
- `sessionVersionSnapshot`,
- timestamps (`authTime`, `lastAccess`),
- optional `sessionIdHint`.

```
#pragma once
#include <cstdint>
#include <optional>
```

```
#include <string>

struct SessionClaims {
  std::int64_t userId{0};
  std::int64_t sessionVersionSnapshot{0};
  std::int64_t authTimeMillis{0};
  std::int64_t lastAccessMillis{0};
  std::string sessionIdHint;

  bool authenticated() const { return userId != 0; }
};

class ISessionClock {
public:
  virtual ~ISessionClock() = default;
  virtual std::int64_t nowMillis() const = 0;
};

struct SessionPolicy {
  std::int64_t idleTimeoutMillis{30LL * 60LL * 1000LL};      // example: 30 minutes
  std::int64_t absoluteTimeoutMillis{24LL * 60LL * 60LL * 1000LL}; // example: 24
  ↪ hours
};
```

ContextProvider: build, validate, refresh

```
#pragma once
#include <string>

enum class ContextStatus {
```

```
  Ok,
  Anonymous,
  Expired,
  Revoked,
  LockedOrDisabled,
  BackendFailure
};

struct ContextResult {
  ContextStatus status{ContextStatus::BackendFailure};
  UserContext ctx;
};

class UserContextProvider {
public:
  UserContextProvider(IUserRepository& users,
                      IPermissionService& perms,
                      ISessionClock& clock,
                      SessionPolicy policy)
    : users_(users), perms_(perms), clock_(clock), policy_(policy) {}

  ContextResult buildFromClaims(const SessionClaims& claims, const std::string&
  ↪  requestId) {
    ContextResult r;
    r.ctx.requestId = requestId;
    r.ctx.sessionIdHint = claims.sessionIdHint;

    if (!claims.authenticated()) {
      r.status = ContextStatus::Anonymous;
```

```
    return r;
  }

  const auto now = clock_.nowMillis();
  if (isExpired(claims, now)) {
    r.status = ContextStatus::Expired;
    return r;
  }

  try {
    const UserSecuritySnapshot snap = users_.loadSecuritySnapshot(claims.userId);

    if (!snap.active || snap.locked) {
      r.status = ContextStatus::LockedOrDisabled;
      return r;
    }
    if (snap.sessionVersion != claims.sessionVersionSnapshot) {
      r.status = ContextStatus::Revoked;
      return r;
    }

    // Construct the context
    r.ctx.userId = snap.userId;
    r.ctx.active = snap.active;
    r.ctx.locked = snap.locked;
    r.ctx.sessionVersion = claims.sessionVersionSnapshot;
    r.ctx.authTimeMillis = claims.authTimeMillis;
    r.ctx.lastAccessMillis = claims.lastAccessMillis;
```

```
      r.ctx.displayName = snap.displayName;
      r.ctx.locale = snap.locale;
      r.ctx.timeZone = snap.timeZone;

      r.ctx.scope = users_.loadScope(snap.userId);
      r.ctx.perms = perms_.loadEffectivePermissions(snap.userId);

      r.status = ContextStatus::Ok;
      return r;

    } catch (...) {
      r.status = ContextStatus::BackendFailure;
      return r;
    }
  }

private:
  IUserRepository& users_;
  IPermissionService& perms_;
  ISessionClock& clock_;
  SessionPolicy policy_;

  bool isExpired(const SessionClaims& c, std::int64_t now) const {
    const auto idleAge = now - c.lastAccessMillis;
    const auto absAge  = now - c.authTimeMillis; // or separate login time if tracked
    if (c.lastAccessMillis == 0 || c.authTimeMillis == 0) return true;
    if (idleAge > policy_.idleTimeoutMillis) return true;
    if (absAge  > policy_.absoluteTimeoutMillis) return true;
    return false;
```

```
  }
};
```

11.4.6 Integrating context provider into the Wt application

A typical Wt application stores `SessionClaims` and updates `lastAccessMillis` at safe boundaries:

- when a request/event begins,
- when a protected page is about to be rendered,
- before a sensitive action.

Session claims storage

```
#pragma once
#include <cstdint>
#include <string>

class SessionState {
public:
  SessionClaims& claims() { return claims_; }
  const SessionClaims& claims() const { return claims_; }

  void onLogin(std::int64_t userId, std::int64_t sessionVersionSnapshot,
               std::int64_t nowMillis, std::string sessionIdHint) {
    claims_.userId = userId;
    claims_.sessionVersionSnapshot = sessionVersionSnapshot;
    claims_.authTimeMillis = nowMillis;
    claims_.lastAccessMillis = nowMillis;
```

```
    claims_.sessionIdHint = std::move(sessionIdHint);
  }

  void touch(std::int64_t nowMillis) { claims_.lastAccessMillis = nowMillis; }

  void onLogout() { claims_ = SessionClaims{}; }

private:
  SessionClaims claims_;
};
```

Obtaining a request id

In production, create a request id for each event to correlate logs/audit. Keep it simple:

```
#include <string>
#include <cstdint>

static std::string makeRequestId(std::int64_t nowMillis, std::int64_t seq) {
  return std::to_string(nowMillis) + "-" + std::to_string(seq);
}
```

11.4.7 Using user context in controllers and pages

The core rule is:

> *Controllers and secured pages should accept a* `UserContext` *(or* `const UserContext&`*) and never query authentication state directly.*

This keeps policy centralized and consistent.

Authorization helpers operating on context

```
#pragma once
#include <stdexcept>
#include <string>

static void requireAuthenticated(const UserContext& ctx) {
  if (!ctx.authenticated()) throw std::runtime_error("Unauthenticated");
  if (!ctx.active || ctx.locked) throw std::runtime_error("Forbidden");
}

static void requirePermission(const UserContext& ctx, const std::string& perm) {
  requireAuthenticated(ctx);
  if (!ctx.perms.has(perm)) throw std::runtime_error("Forbidden");
}

static void requireTenantScope(const UserContext& ctx, std::int64_t
↪  requiredTenantId) {
  requireAuthenticated(ctx);
  if (!ctx.scope.tenantId.has_value() || *ctx.scope.tenantId != requiredTenantId) {
    throw std::runtime_error("Forbidden");
  }
}
```

Controller method signature

```
#pragma once
#include <string>

class AdminUsersController {
```

```
public:
  std::string listUsers(const UserContext& ctx) {
    requirePermission(ctx, "user.read.any");
    // Fetch users from repository and return serialized representation or build
    ↪  view model.
    return "OK";
  }

  void disableUser(const UserContext& ctx, std::int64_t targetUserId) {
    requirePermission(ctx, "user.disable");
    (void)targetUserId;
    // Perform transaction; emit audit; handle errors.
  }
};
```

11.4.8 Context refresh and invalidation strategies

Caching permissions and security snapshot improves performance, but it must remain correct. Use one (or both) of these strategies:

Strategy A: Session version revocation

- Keep `sessionVersion` on the user record.
- Increment it on password reset, role changes, admin revocation.
- Context provider compares claim snapshot vs current DB version.

This is the strongest and simplest global revocation mechanism.

Strategy B: Time-based refresh

- Refresh permissions/security snapshot periodically (e.g., every N minutes).
- Use this when policy changes must be reflected quickly but global revocation is not available.

If you use refresh, keep it explicit and bounded to avoid DB load spikes.

Explicit invalidation hooks

When the current session changes roles/permissions (or an admin changes them), invalidate immediately:

- clear cached permissions in session,
- force context rebuild on next request,
- optionally redirect to a safe page if permissions are reduced.

11.4.9 Step-up authentication support

For sensitive actions (role assignment, password change, exporting audit logs), require recent authentication.

Step-up gate

```
#include <cstdint>
#include <stdexcept>

static void requireRecentAuth(const UserContext& ctx,
                              std::int64_t nowMillis,
                              std::int64_t maxAgeMillis)
```

```
{
  requireAuthenticated(ctx);
  const auto age = nowMillis - ctx.authTimeMillis;
  if (age > maxAgeMillis) {
    throw std::runtime_error("StepUpRequired");
  }
}
```

11.4.10 Administrative impersonation (optional, high risk)

If you support “impersonate user”:

- represent **two identities** in context:
  - **actor** (admin identity),
  - **effective user** (impersonated).
- enforce a dedicated permission such as `admin.impersonate`,
- record detailed audit events for every impersonated action,
- provide a visible UI indicator and an immediate exit option.

Impersonation-aware context sketch

```
struct Impersonation {
  std::int64_t actorUserId{0};      // admin
  std::int64_t effectiveUserId{0};  // target
  bool active{false};
};

struct UserContext2 : UserContext {
```

```
  std::optional<Impersonation> impersonation;
};
```

If you do not need impersonation, do not implement it; it increases security risk and audit requirements.

11.4.11 Operational checklist

- Store only minimal trusted claims in session; rebuild full context via provider.
- Validate session version, lock/disable, and timeouts before accessing secured pages.
- Cache permissions for fast checks; invalidate deterministically on policy changes.
- Use a uniform `UserContext` passed to controllers and pages.
- Implement step-up gating for sensitive operations.
- Never trust UI visibility; enforce permissions at execution.
- Log/audit using request id and safe identifiers; never log secrets.

11.4.12 Summary

User context management is the glue between authentication, authorization, and application behavior. A production-grade Wt UMS should:

- maintain compact session claims,
- construct a validated `UserContext` through a dedicated provider,
- centralize permission and scope checks via helpers,
- support revocation, timeouts, and step-up authentication,

- keep controllers and pages free of ad-hoc session lookups.

This yields a security model that remains consistent, testable, and maintainable as the system grows.

Part VI

Database and Data Layer

Introduction to Wt::Dbo

12.1 ORM philosophy in Wt

ObjectRelational Mapping (ORM) is a design pattern that bridges the gap between object-oriented programming languages (such as C++) and relational databases. In the context of Wt applications, the ORM serves these crucial purposes:

- **Productivity**: reduce boilerplate code for queries, joins, and mapping between tables and C++ classes.
- **Safety**: encourage type-safe access to persistent entities and relationships.
- **Maintainability**: centralize schema definitions and access patterns within a unified C++ domain model.
- **Consistency**: ensure that object graphs and database transactions behave in a predictable and correct fashion.
- **Separation of concerns**: allow the data layer to evolve independently of business logic and presentation.

Wt provides `Wt::Dbo`, a lightweight, header-only ORM that integrates cleanly with modern C++ and supports multiple backends (PostgreSQL, MySQL, SQLite, MSSQL). `Wt::Dbo` is not

a heavy meta-programming framework, but rather a practical toolkit with the following philosophy:

- **Explicit mapping**: every persistent class must explicitly define its mapping to the database schema.
- **Composability**: relationships (one-to-one, one-to-many, many-to-many) are explicit and navigable.
- **Transactional integrity**: all modifications are scoped to a transaction object that must be committed explicitly, reducing implicit side effects.
- **Lazy loading with clear control**: related objects can be fetched on demand, but the programmer controls fetch depth and timing to avoid performance surprises.
- **Backend abstraction**: while SQL dialects differ, `Wt::Dbo` abstracts differences while allowing backend-specific tuning where necessary.
- **No global state**: mapping and session state are encapsulated in session objects, reducing coupling and improving testability.

12.1.1 Core concepts in Wt::Dbo

`Wt::Dbo` works with a few fundamental concepts:

Session

A `dbo::Session` represents the connection to a database. It holds a map of classes to database tables and manages identity maps, cached pointers, and lazy-loading behavior.

Transaction

All queries and modifications are performed within a `dbo::Transaction`. Transactions provide atomicity and isolation. They also serve as a context for object identity resolution.

Persistent objects

Persistent C++ classes are normal structs/classes extended with fields and relationships that are mapped to table columns and joins.

Pointers

`dbo::ptr<T>` and `dbo::weak_ptr<T>` represent persistent object references. These pointers may refer to already-loaded or yet-to-be-fetched entities.

Mapping

Mapping is the act of registering C++ classes with the session and defining how their members correspond to database constructs (columns, join tables, etc.).

12.1.2 Mapping philosophy

A robust ORM mapping in Wt::Dbo adheres to:

- **Explicit registration**: developers must call `session.mapClass<T>()` for each persistent entity.
- **Field-by-field mapping**: members are mapped via `dbo::field()`, which enforces compile-time consistency between C++ types and database column types.
- **Relation clarity**: relations (one-to-many, many-to-many) are declared with clear join table names and foreign keys.

- **Join table control**: many-to-many relations explicitly define join table names, reducing ambiguity and enabling schema version control.

This philosophy contrasts with "convention over configuration" ORMs that auto-generate schema and field mappings. Wt::Dbo favors deliberate mapping so that C++ types and database schemas evolve together under programmer control.

12.1.3 Transactions and identity

Transactions in `Wt::Dbo` serve dual purposes:

- **Atomicity**: all reads and writes in a transaction are isolated and committed together.
- **Identity map**: within a transaction, the same database row maps to the same C++ object instance, preserving pointer identity and reducing unnecessary fetches.

A typical transaction scope looks like:

```
#include <Wt/Dbo/Transaction.h>

void doSomething(dbo::Session& sess) {
  dbo::Transaction t(sess);
  // queries and modifications
  t.commit();
}
```

If `commit()` is not called, all modifications are discarded at the end of the scope.

12.1.4 Lazy loading and eager fetch

By default, related objects are loaded lazily. This means:

- **Performance control**: developers decide when to fetch related entities to avoid N+1 query problems.
- **Clarity**: `dbo::ptr<T>::load()` and `dbo::collection<T>::load()` make explicit the fetch boundaries.

Example:

```
auto user = sess.find<User>().where("email = ?").bind(email).resultValue();
if (user) {
  user->roles.load(); // explicit eager fetch of related roles
}
```

Developers should balance lazy and eager fetches based on expected access patterns.

12.1.5 Change tracking

Wt::Dbo tracks changes to persistent objects in a transaction:

- modifications to fields and relations are detected within the active transaction,
- on `commit()`, changes are flushed to the database in a deterministic order.

This model avoids “hidden writes” and makes debugging state transitions easier.

12.1.6 Polymorphism support

Wt::Dbo supports class hierarchies via single-table inheritance or table-per-class patterns. Polymorphic mapping allows:

- storing derived classes in a normalized schema,
- retrieving base-class pointers that refer to the correct derived class,

- dynamic casting via `dbo::ptr<Derived>`.

This is important in domains where entities share common behavior (e.g., different user types).

12.1.7 Query mechanisms

Wt::Dbo provides multiple ways to query:

- **Criteria-based queries**: `where()` and `bind()` constructs for parameterized SQL.
- **Result sets**: iterate over collections of persistent pointers.
- **Aggregations and raw SQL**: for performance-critical cases, developers may embed raw SQL with safety considerations.

Example of a query:

```
auto users = sess.find<User>()
                 .where("active = ?")
                 .bind(true)
                 .orderBy("email")
                 .resultList();
for (auto& u : users) {
  // use u
}
```

12.1.8 Schema evolution and migrations

Wt::Dbo can create tables automatically with `session.createTables()`, but in production systems developers typically use versioned migrations:

- define migrations in C++ or SQL,

- version the schema on disk (migration scripts),
- apply migrations selectively based on an applied version marker.

Automated `createTables()` is useful in development but inadequate for controlled evolution of production databases.

12.1.9 Error handling in ORM operations

ORM operations may fail due to:

- constraint violations (unique keys, foreign keys),
- connection failures,
- type mismatches or unexpected nulls,
- transaction conflicts.

Robust applications must:

- catch exceptions thrown by `dbo::Transaction` and underlying drivers,
- distinguish between expected constraint errors and system failures,
- surface generic errors to users while recording detailed diagnostics for logs/audit.

12.1.10 Performance considerations

ORMs introduce abstraction overhead. To mitigate:

- use eager fetch judiciously to avoid N+1 queries,
- cache lookup results when possible (e.g., permission sets),

- prefer bulk operations in transactions where safe,
- profile database calls and optimize hot paths with indices and tuned SQL.

12.1.11 Testing and testability

Wt::Dbo encourages testable designs:

- session and backend can be swapped (SQLite for tests, PostgreSQL in production),
- transactions can be rolled back for repeatable tests,
- repository abstractions wrap Dbo calls for unit testing.

12.1.12 Integration with application architecture

In a layered architecture:

- the ORM resides in the **data layer**,
- domain services (business logic) depend on repository interfaces,
- controllers and UI builders depend on domain services,
- ORM specifics remain isolated from higher layers.

This separation enables evolving the data model independent of UI and business logic.

12.1.13 Summary

Wt::Dbo's ORM philosophy is pragmatic and C++-centric. It emphasizes:

- explicit mapping of persistent classes,

- clear transaction boundaries,
- identity and relationship management,
- performance-aware loading strategies,
- testability and maintainability.

For production-grade applications, developers should combine ORM capabilities with disciplined schema migrations, robust error handling, and optimized query planning. This yields a stable and evolvable data layer for Modern C++ Web Development with Wt.

12.2 Sessions and transactions

In `Wt::Dbo`, the two central runtime concepts are **sessions** and **transactions**. They define how your application:

- connects to a database backend,
- maps C++ classes to relational tables,
- executes queries safely,
- persists changes atomically,
- manages object identity and lazy-loading behavior.

A production-grade data layer must treat sessions and transactions as **first-class architectural boundaries**. Most reliability and data-consistency bugs in ORM-backed systems come from unclear ownership of these boundaries.

12.2.1 The role of `dbo::Session`

What a session represents

A `dbo::Session` is the primary ORM context. It encapsulates:

- the **database connection** object (via a backend connection),
- the **mapping registry** of C++ classes to tables,
- runtime metadata used by the ORM (identity map, SQL generation, type conversion),
- the context for persistent pointer resolution (`dbo::ptr<T>`).

You should think of `dbo::Session` as *the unit that binds database access to the mapping model*. It is not a mere connection handle.

Session responsibilities

A `dbo::Session` is responsible for:

- registering all persistent classes (`mapClass<T>(...)`) once during initialization,
- holding the backend connection (e.g., PostgreSQL, MySQL, SQLite, MSSQL),
- providing factory methods for queries (`find<T>()`, `query<...>()`),
- creating and managing table schema (development mode: `createTables()`),
- coordinating object identity and persistence behavior during transactions.

Session lifetime and ownership

In a Wt application, you typically have two acceptable patterns:

Pattern A: Session per application session (per user session) Each Wt application instance maintains its own `dbo::Session` object:

- simple and ergonomic,
- works well with SQLite or low concurrency,
- may not scale well under heavy concurrent load without connection pooling.

Pattern B: Session from a pool (recommended for production) The application obtains a session (or a backend connection) from a pool:

- reduces connection churn,
- fits multi-threaded web server scaling,
- supports high traffic and multiple Wt server processes.

The correct choice depends on deployment architecture, database backend, and expected concurrency.

12.2.2 The role of `dbo::Transaction`

What a transaction represents

A `dbo::Transaction` is a scope guard that begins a database transaction on construction and closes it on destruction. It is the correct boundary for:

- atomic reads and writes,
- consistent snapshot semantics (depending on database isolation level),
- ORM change tracking and flushing,

- stable identity resolution of persistent objects.

The transaction is a **required context** for safe modifications and for many ORM operations.

Commit vs rollback semantics

- If you call `commit()`, changes are flushed and persisted.
- If you do not call `commit()`, the transaction rolls back on destruction.

This rule makes transactions safe by default: failures do not silently persist partial updates.

Transaction scope is a correctness boundary

A transaction must not be "accidentally long":

- Do not keep a transaction open across network calls, email sending, or long computations.
- Do not keep a transaction open across multiple UI interactions.
- Do not keep a transaction open while waiting for user input.

Transactions should be short and tightly scoped around database operations.

12.2.3 Minimal example: creating a session and running a transaction

```
#include <Wt/Dbo/Dbo.h>
#include <Wt/Dbo/backend/Sqlite3.h>
#include <Wt/Dbo/Transaction.h>

namespace dbo = Wt::Dbo;
```

```
struct User {
  std::string email;
  bool active{true};

  template <class Action>
  void persist(Action& a) {
    dbo::field(a, email, "email");
    dbo::field(a, active, "active");
  }
};

int main() {
  dbo::backend::Sqlite3 sqlite("app.db");
  dbo::Session session;
  session.setConnection(sqlite);

  session.mapClass<User>("users");

  {
    dbo::Transaction t(session);
    session.createTables(); // development convenience
    t.commit();
  }

  {
    dbo::Transaction t(session);
    dbo::ptr<User> u = session.add(std::make_unique<User>(User{"user@example.com",
    ↪  true}));
    t.commit();
```

```
  }

  return 0;
}
```

12.2.4 Why transactions matter even for reads

Many systems assume transactions are only for writes. In practice, transactions are important for reads too:

- consistent multi-step reads (read user + roles + permissions),
- stable object identity (the same row maps to the same object within the transaction),
- predictable lazy-loading behavior when traversing relations.

For example, if your code loads a user and then iterates roles, a transaction ensures that any lazy loads occur safely inside a consistent database context.

12.2.5 Identity map and object stability

A key ORM feature is that within a transaction:

- two fetches of the same row typically refer to the same `dbo::ptr<T>` identity,
- updates to the object are tracked and written on commit,
- relations can be traversed and loaded without duplicating object instances.

This property matters for business logic:

- it prevents subtle bugs where you update one instance but read another,
- it enables safe navigation of object graphs.

12.2.6 Transactions and lazy loading

Wt::Dbo can load related objects lazily. Lazy loading is convenient, but it must obey a strict rule:

If you will traverse relations or call `load()`, do so inside an active transaction.

Example: explicitly loading roles inside the transaction:

```
dbo::Transaction t(session);

dbo::ptr<User> u =
  session.find<User>().where("email = ?").bind("user@example.com");

if (u) {
  // If roles is a collection relation:
  u->roles.load();   // safe within transaction
}

t.commit();
```

If you attempt to lazy-load outside of a transaction, behavior can become invalid or backend-dependent.

12.2.7 Nested transactions and composition

Database backends differ in how they support nested transactions. A safe design principle is:

- avoid implicit nested transactions across multiple layers,
- pass a transaction context through the call chain when needed,

- use a single outer transaction for a single unit of work.

In practice:

- Controllers start transactions at the boundary of a use-case.
- Repository and service functions assume they run inside that transaction.

This avoids deadlocks, partial commits, and inconsistent reads.

12.2.8 Unit of work pattern

Transactions naturally express the **unit of work** pattern:

- load the objects needed,
- modify them in memory,
- commit to persist all changes atomically.

Example: disable a user and revoke sessions as one unit:

```
void disableUser(dbo::Session& session, dbo::ptr<User> user) {
  dbo::Transaction t(session);

  user.modify()->active = false;

  // Additional updates can happen here: revoke sessions, log audit records, etc.

  t.commit();
}
```

If anything fails, the whole unit rolls back.

12.2.9 Isolation levels and practical expectations

Relational databases implement different isolation levels (Read Committed, Repeatable Read, Serializable). Your ORM code must be written to be correct under the chosen isolation. Practical guidelines:

- keep transactions short to reduce contention,
- use appropriate indexes so queries do not lock excessively,
- prefer explicit unique constraints and handle conflicts deterministically,
- avoid reading then writing based on stale assumptions when high concurrency is expected.

If you need strict invariants (no duplicate usernames, consistent role assignment), enforce them via:

- unique constraints,
- foreign keys,
- carefully designed transactions that update related rows together.

12.2.10 Error handling patterns

Operations may fail due to:

- constraint violations,
- connection problems,
- timeouts,

- serialization failures (in high isolation levels).

A production pattern is to classify errors:

- **expected domain errors**: e.g., unique constraint violation,
- **transient errors**: retryable (timeouts, serialization conflicts),
- **system errors**: not retryable without operator intervention.

Wrap transactions with controlled exception handling at the controller boundary:

```
template <typename Fn>
bool withTransaction(dbo::Session& session, Fn&& fn) {
  dbo::Transaction t(session);
  try {
    fn();
    t.commit();
    return true;
  } catch (...) {
    // Transaction will roll back automatically.
    return false;
  }
}
```

In production, you typically log exceptions and map them to safe UI outcomes.

12.2.11 Concurrency and session usage constraints

A critical operational rule:

> **Do not share one `dbo::Session` instance across threads unless you can guarantee thread safety and backend behavior.**

Instead:

- create separate sessions per thread,
- or use a connection/session pool to allocate a backend connection per request/unit of work.

This aligns with typical web server concurrency: each request or event handler uses its own short-lived transaction.

12.2.12 Practical recipes for Wt applications

Recipe 1: Transaction per use-case

For every controller method that modifies the database:

- start one transaction,
- call repository/service methods,
- commit once at the end.

Recipe 2: Keep UI logic outside transactions

Do not hold a transaction while:

- rendering large UI trees,
- waiting for user events,
- calling remote APIs.

Instead:

- load needed data into a view model,

- close transaction,
- render the UI using the view model.

Recipe 3: Explicit load boundaries

If you require related objects, load them inside a transaction:

- call `load()` explicitly,
- avoid accidental lazy loads scattered across code.

12.2.13 Summary

In `Wt::Dbo`, sessions and transactions define the safety and correctness of your entire data layer:

- **`dbo::Session`** owns the mapping model and database connection context.
- **`dbo::Transaction`** defines atomicity, consistency, and the unit-of-work boundary.
- Use transactions for both reads and writes when you need consistent object graphs and safe lazy loading.
- Keep transactions short, avoid nested transaction ambiguity, and enforce concurrency-safe session usage.
- Centralize transaction control at use-case boundaries for predictable and maintainable systems.

These principles ensure that `Wt::Dbo` remains a reliable foundation for a production-grade data layer in modern C++ Wt applications.

12.3 Object-relational mapping

Object-relational mapping (ORM) is a core data architecture pattern for modern applications that must bridge between an object-oriented language (C++) and a relational database system. Wts ORM subsystem, `Wt::Dbo`, provides a practical and C++-centric approach to persistence that supports robust applications, manageable domain models, and efficient database interaction.
In this section, we present the philosophy, structure, and best practices for ORM using `Wt::Dbo` in a production context. The goal is to equip developers with a deep understanding of how mapping works, how to design persistent classes, and how to maintain clarity, performance, and data integrity in real-world systems.

12.3.1 The need for ORM

Relational databases and C++ objects have different models:

- Relational data is normalized into tables, rows, and foreign keys.
- C++ data is structured in classes, inheritance, pointers, and containers.
- Manual SQL mixing in business logic leads to:
  - boilerplate code,
  - error-prone conversions,
  - inconsistent naming and type mismatches,
  - duplicated query fragments.

ORM provides a principled way to unify these models so that:

- C++ classes represent database entities directly,

- relationships are navigable through language constructs,
- schema evolution can be managed with versioned migrations,
- transactions encapsulate consistent units of work,
- type safety is enforced at compile time and runtime.

12.3.2 Mapping domain classes to tables

Persistent classes in `Wt::Dbo` are C++ types augmented with a `persist` method. This method maps class members (data and relations) to relational constructs.

Minimal persistent class

The simplest mapping starts with a struct and a `persist()` method:

```
#pragma once
#include <string>
#include <Wt/Dbo/Dbo.h>

namespace dbo = Wt::Dbo;

class User {
public:
  std::string email;
  bool active{true};

  template <typename Action>
  void persist(Action& a) {
    dbo::field(a, email, "email");
```

```
    dbo::field(a, active, "active");
  }
};
```

The `persist()` method is called by the ORM to understand how fields map to database columns.

Mapping classes to the session

Before using a persistent class, the session must register it:

```
dbo::Session session;
session.setConnection(std::make_unique<dbo::backend::Postgres>("conninfo"));
session.mapClass<User>("users");
```

This mapping ties the C++ class `User` to the database table `users`.

12.3.3 Mapping object fields

`dbo::field()` maps a simple member to a column. It enforces:

- C++ type to SQL type conversion,
- non-nullability when applicable,
- consistent naming with schema or generated tables.

You can customize column names and constraints via additional traits or backend details.

12.3.4 Mapping relationships

Relationships between entities are essential in domain modeling. `Wt::Dbo` supports:

- **One-to-many**,
- **Many-to-many**,
- **One-to-one** (through unique pointers),
- **Bidirectional navigation**.

One-to-many relationship

Suppose each `User` has many `Post` objects:

```
class Post {
public:
  std::string title;
  std::string body;

  dbo::ptr<User> author;

  template <typename Action>
  void persist(Action& a) {
    dbo::field(a, title, "title");
    dbo::field(a, body,  "body");
    dbo::belongsTo(a, author, "author_id");
  }
};

class User {
```

```
public:
  std::string email;
  dbo::collection< dbo::ptr<Post> > posts;

  template <typename Action>
  void persist(Action& a) {
    dbo::field(a, email, "email");
    dbo::hasMany(a, posts, dbo::ManyToOne, "author_id");
  }
};
```

This establishes a foreign key with the `author_id` column and allows navigating from `User` to associated `Post` objects.

Many-to-many relationship

For many-to-many, a join table is explicitly declared:

```
class Role;
class User;

class Permission {
public:
  std::string key;

  dbo::collection< dbo::ptr<Role> > roles;

  template <typename Action>
  void persist(Action& a) {
    dbo::field(a, key, "key");
```

```
    dbo::hasMany(a, roles, dbo::ManyToMany, "role_permissions",
                 "permission_id", "role_id");
  }
};
```

Here, the join table `role_permissions` links `Permission` and `Role`.

12.3.5 Lazy loading and explicit fetch control

By default, `dbo::ptr<T>` and `dbo::collection<T>` do not fetch related objects until explicitly asked. This is deliberate:

- avoid unbounded eager loading,
- keep transactions short and predictable,
- allow developers to control fetch boundaries.

To fetch related data, call `load()`:

```
user->posts.load(); // explicitly fetch posts inside a transaction
```

Failure to fetch inside a transaction can lead to undefined behavior or backend-specific errors.

12.3.6 Object identity and caching

During an active transaction:

- retrieving the same row twice yields the same `dbo::ptr<T>` identity,
- modifying an object updates it in-memory and schedules an update on commit,
- cross-references are maintained consistently.

This identity map prevents duplicated objects and inconsistent state within a transaction.

12.3.7 Transactions and change tracking

Wt::Dbo tracks changes only within the context of a `dbo::Transaction`. For example:

```
dbo::Transaction t(session);

auto u = session.find<User>().where("id = ?").bind(1);
u.modify()->email = "new@example.com";

// Related changes also tracked here...

t.commit(); // flushes all changes
```

If `commit()` is omitted, all changes are rolled back.

12.3.8 Polymorphism and inheritance

Wt::Dbo supports class hierarchies:

- single-table inheritance,
- table-per-class strategies.

Persistent pointers handle downcasts safely via `dbo::ptr<Derived>` when the underlying row corresponds to a derived class.

12.3.9 Query API

Wt::Dbo provides a fluent query API:

```
auto users = session.find<User>()
                .where("active = ?").bind(true)
```

```
                .orderBy("email")
                .resultList();

for (auto& u : users) {
  // use u
}
```

Queries can be parameterized and composed using `where()`, `bind()`, `orderBy()`, and result-set iteration.

12.3.10 Error handling in ORM operations

ORM operations include:

- constraint violations,
- connection errors,
- transaction conflicts,
- relation-load failures.

Production code should:

- catch exceptions at the transaction boundary,
- categorize domain errors vs system errors,
- return generic messages to the UI,
- log detailed diagnostics internally.

12.3.11 Schema evolution and migrations

While `session.createTables()` can bootstrap schema, production systems require:

- versioned migrations,
- backward-compatible changes,
- rollback scripts,
- controlled execution in deployment pipelines.

Explicit migrations reduce risks of data loss and inconsistent production schemas.

12.3.12 Performance considerations

Mapper strategies must consider:

- eager vs lazy loading balance,
- indices to support query patterns,
- batch operations when updating many rows,
- minimizing transaction scope.

12.3.13 Separation of concerns in ORM

Good architecture isolates the ORM:

- repository classes encapsulate Dbo usage,
- domain services use repositories,
- controllers orchestrate use cases with services,
- UI remains free of persistence internals.

12.3.14 Testing with ORM

Testing with Dbo is practical if you:

- use in-memory backends (SQLite) for unit tests,
- reset schema between test runs,
- wrap transactions and rollback for data isolation.

12.3.15 Summary

Object-relational mapping in `Wt::Dbo` provides:

- explicit, type-safe mapping of C++ classes to relational data,
- clear relationship modeling,
- controlled lazy loading,
- transaction-aware persistence,
- identity and navigation support,
- production-ready patterns for error handling, migrations, and separation of concerns.

Following these principles yields a maintainable and performant data layer for server-side applications built with Wt.

Data Modeling

13.1 User model

The **User model** is the most security-critical entity in the User Management System (UMS). It is both a persistence model (database schema) and a domain model (C++ object used by business logic). A production-grade user model must satisfy:

- **Identity correctness**: stable primary key, unique login identifiers, and canonicalization rules.
- **Security**: safe password storage (hash only), lockout controls, and session revocation support.
- **Operational needs**: audit-friendly timestamps, soft-delete or status flags, and migration stability.
- **Authorization integration**: links to roles/permissions and support for effective permission derivation.
- **Internationalization**: correct handling of display names, locale, and timezone.

This section defines an academically rigorous user model design, including schema decisions, constraints, indexing strategy, and a complete `Wt::Dbo` mapping.

13.1.1 Principles of a production user model

1) Separate identity from credentials

- Identity: `id`, profile, status.
- Credentials: password hash, password change timestamps, MFA secrets (if any).

You may store credentials in the same table for simplicity, but the conceptual separation must remain clear in your design and code.

2) Unique identifiers and canonicalization

A user typically authenticates with one of:

- email,
- username,
- phone number,
- external identity (OIDC subject).

For a classic email-based login:

- store a canonical email form (e.g., trimmed, lowercased as policy),
- enforce uniqueness on the canonical value,
- do not reveal existence through UI messages.

3) Store passwords safely

The user model must never store:

- plain passwords,
- reversible encryption of passwords,
- unsalted hashes.

Instead store only:

- password hash (from a slow, salted password hashing algorithm),
- a password version or timestamp (for session revocation and policy checks).

4) Built-in session revocation

To support forced logout and revocation across devices, keep a monotonic field such as:

- `session_version` (integer),
- or `credentials_version` (integer),
- or `password_changed_at` (timestamp).

Increment/update it when:

- password changes,
- roles/permissions change for the user,
- account is locked/disabled,
- administrator forces logout.

5) Status fields instead of hard deletes

Hard deleting users is usually harmful to auditability. Prefer:

- `active` flag,
- `locked` flag,
- optional `deleted_at` timestamp (soft delete).

13.1.2 Schema design

A minimal but production-oriented schema for `users`:

- `id` (primary key, auto-increment or UUID depending on backend policy),
- `email` (canonical, unique),
- `display_name` (for UI),
- `password_hash` (string/blob),
- `active` (boolean),
- `locked` (boolean),
- `failed_login_count` (integer),
- `lock_until` (timestamp or integer epoch millis),
- `session_version` (integer),
- `created_at`, `updated_at`, `last_login_at` (timestamps),
- `locale`, `time_zone` (strings).

Indices and constraints

- unique index on `email`,
- index on `active` if you frequently query active users,
- index on `locked` or `lock_until` if you run cleanup/unlock jobs.

13.1.3 C++ domain model and `Wt::Dbo` mapping

Below is a complete `User` entity model designed for `Wt::Dbo`. It includes:

- field mapping,
- safe defaults,
- timestamps,
- session version support,
- hooks for relations (roles, sessions) defined later in the chapter.

```
#pragma once

#include <string>
#include <cstdint>
#include <chrono>
#include <optional>

#include <Wt/Dbo/Dbo.h>
#include <Wt/Dbo/Types.h>

namespace dbo = Wt::Dbo;
```

```cpp
static inline std::int64_t nowMillisUtc() {
  using namespace std::chrono;
  return
  ↪  duration_cast<milliseconds>(system_clock::now().time_since_epoch()).count();
}

class Role;          // forward: many-to-many (User <-> Role)
class SessionRecord; // forward: one-to-many (User -> SessionRecord)

class User {
public:
  // --- Identity / login identifiers ---
  std::string email;           // canonical email used for login (unique)
  std::string displayName;     // UI display name (not a security identifier)

  // --- Credentials ---
  std::string passwordHash;    // store only the hash output, never plaintext

  // --- Account state ---
  bool active{true};
  bool locked{false};

  std::int32_t failedLoginCount{0};
  std::optional<std::int64_t> lockUntilMillis; // lockout expiry (UTC epoch millis)

  // --- Session revocation / forced logout ---
  std::int64_t sessionVersion{0};  // increment to revoke all sessions
```

```
// --- Timestamps (UTC epoch millis for portability across DB backends) ---
std::int64_t createdAtMillis{0};
std::int64_t updatedAtMillis{0};
std::optional<std::int64_t> lastLoginAtMillis;

// --- Preferences (non-security, but stored here for convenience) ---
std::string locale{"en"};
std::string timeZone{"UTC"};

// --- Relations (declared now, fully used later) ---
dbo::collection< dbo::ptr<Role> > roles;
dbo::collection< dbo::ptr<SessionRecord> > sessions;

User() = default;

explicit User(std::string canonicalEmail,
              std::string name,
              std::string pwHash)
  : email(std::move(canonicalEmail)),
    displayName(std::move(name)),
    passwordHash(std::move(pwHash))
{
  const auto now = nowMillisUtc();
  createdAtMillis = now;
  updatedAtMillis = now;
}

bool isLockActive(std::int64_t nowMillis) const {
  if (!locked) return false;
```

```
  if (!lockUntilMillis.has_value()) return true; // locked indefinitely
  return nowMillis < *lockUntilMillis;
}

void touchUpdated() { updatedAtMillis = nowMillisUtc(); }

template <typename Action>
void persist(Action& a) {
  dbo::field(a, email,          "email");
  dbo::field(a, displayName,    "display_name");

  dbo::field(a, passwordHash,   "password_hash");

  dbo::field(a, active,         "active");
  dbo::field(a, locked,         "locked");

  dbo::field(a, failedLoginCount, "failed_login_count");
  dbo::field(a, lockUntilMillis,  "lock_until_millis");

  dbo::field(a, sessionVersion, "session_version");

  dbo::field(a, createdAtMillis, "created_at_millis");
  dbo::field(a, updatedAtMillis, "updated_at_millis");
  dbo::field(a, lastLoginAtMillis, "last_login_at_millis");

  dbo::field(a, locale,         "locale");
  dbo::field(a, timeZone,       "time_zone");

  // Relations:
```

```
    // roles: many-to-many, join table name selected later.
    // sessions: one-to-many, foreign key selected later.
    dbo::hasMany(a, roles, dbo::ManyToMany, "user_roles",
                 "user_id", "role_id");

    dbo::hasMany(a, sessions, dbo::ManyToOne, "user_id");
  }
};
```

13.1.4 Mapping and constraints setup

Define mapping during application initialization. Example:

```
#include <Wt/Dbo/Session.h>

static void mapEntities(dbo::Session& s) {
  s.mapClass<User>("users");
  // Later in this chapter:
  // s.mapClass<Role>("roles");
  // s.mapClass<SessionRecord>("sessions");
}
```

For production, enforce unique constraints with schema migrations. If you use `createTables()` for development, still plan migrations for production.

13.1.5 User model behaviors used by authentication

The user entity supports authentication policy logic:

Failed login handling and lockout

- increment `failedLoginCount` on each failure,

- set `locked` and `lockUntilMillis` when threshold exceeded,
- reset count on successful login,
- update `lastLoginAtMillis` on success.

A policy helper (domain service) should implement these updates, not the UI.

Session revocation on security events

- increment `sessionVersion` on password change,
- increment `sessionVersion` on role assignment changes,
- increment `sessionVersion` on admin forced logout.

The session subsystem compares the session's stored snapshot to `User::sessionVersion`.

13.1.6 Production recommendations

Avoid mixing profile and security claims excessively

If your application grows, split into:

- `users` (security-critical fields),
- `user_profiles` (optional preferences).

Prefer UTC time representation

Store timestamps in UTC. Choose a representation:

- DB native timestamps with timezone support (if consistent across backends),
- or epoch millis (portable and simple across multiple backends).

Design for migrations

Even if you bootstrap tables with `createTables()` during development:

- maintain a migration system for production,
- version your schema and apply controlled changes.

13.1.7 Summary

The user model in a production Wt application must be designed as a security boundary, not just a data container. The essential characteristics are:

- stable identity and unique login identifier,
- safe credential storage (hash only),
- lockout and state flags,
- session revocation support via `sessionVersion`,
- audit-friendly timestamps,
- clean integration with roles, permissions, and session records.

This model becomes the foundation for the rest of Chapter 13: roles, permissions, session tables, and advanced indexing strategies.

13.2 Role model

The **Role model** is the backbone of authorization in the User Management System (UMS). While permissions express fine-grained capabilities (“can do X”), roles provide a manageable abstraction that groups permissions and assigns them to users.
In production systems, a correct role model must satisfy:

- **Clarity**: roles are stable identifiers for policy bundles.
- **Least privilege**: users get only the roles necessary for their tasks.
- **Auditability**: role assignments and changes are traceable.
- **Scalability**: role evaluation remains efficient at runtime.
- **Migrations-friendly**: role keys do not break when names change.

This section defines a production-grade role schema, constraints, indexing, and a complete `Wt::Dbo` mapping suitable for the live project.

13.2.1 Role semantics and boundaries

Role as a policy bundle

A role is a named set of permissions. Typical examples:

- `admin`: manage users, roles, and system settings
- `manager`: manage a subset of business objects
- `user`: access own profile and personal resources

Role identifier vs role display name

Never use a human-facing name as your policy key. Use:

- **role key** (stable, unique): `"admin"`, `"support"`, `"billing_manager"`
- **display name** (localizable): `"Administrator"`, `"Support Agent"`

Global vs scoped roles

For multi-tenant systems, roles may be:

- **global roles**: apply across the entire system,
- **tenant-scoped roles**: apply within a tenant or organization boundary.

In this live project, we model global roles first, and we keep the schema extensible for scoped roles later.

13.2.2 Schema design

A minimal, robust `roles` table should include:

- `id` (primary key),
- `key` (unique, stable role identifier),
- `display_name` (human-readable label),
- `description` (optional, for administration UI),
- `system` (boolean: protected built-in roles),
- `created_at`, `updated_at` (timestamps).

Constraints and indices

- unique index on `key`,
- optional index on `system` for admin listings/filtering,
- ensure `key` validation policy (length, allowed chars) at the application layer.

13.2.3 Many-to-many relationships

Roles participate in two critical many-to-many relationships:

- **User** ↔ **Role** via `user_roles`
- **Role** ↔ **Permission** via `role_permissions`

This design provides:

- flexible role assignments,
- permission reuse across roles,
- clean join tables for auditing and migrations.

13.2.4 C++ domain model and `Wt::Dbo` mapping

Below is a production-grade `Role` entity suitable for `Wt::Dbo`. It includes:

- stable role key,
- display metadata,
- system protection flag,
- timestamps,
- relation collections for users and permissions.

```
#pragma once

#include <string>
#include <cstdint>
#include <chrono>
```

```
#include <Wt/Dbo/Dbo.h>
#include <Wt/Dbo/Types.h>

namespace dbo = Wt::Dbo;

static inline std::int64_t nowMillisUtc() {
  using namespace std::chrono;
  return std::chrono::duration_cast<std::chrono::milliseconds>(
           std::chrono::system_clock::now().time_since_epoch()
         ).count();
}

class User;        // forward: many-to-many
class Permission;  // forward: many-to-many

class Role {
public:
  // Stable role identifier used in code and migrations (unique)
  std::string key;               // e.g. "admin", "user", "support"

  // Human-readable metadata (can be localized at the UI layer)
  std::string displayName;       // e.g. "Administrator"
  std::string description;       // optional

  // Protected built-in roles (cannot be removed from production easily)
  bool system{false};

  // Timestamps (UTC epoch millis)
```

```
std::int64_t createdAtMillis{0};
std::int64_t updatedAtMillis{0};

// Relations:
dbo::collection< dbo::ptr<User> > users;               // User <-> Role
dbo::collection< dbo::ptr<Permission> > permissions;   // Role <-> Permission

Role() = default;

Role(std::string roleKey,
     std::string roleDisplayName,
     std::string roleDescription,
     bool isSystem = false)
  : key(std::move(roleKey)),
    displayName(std::move(roleDisplayName)),
    description(std::move(roleDescription)),
    system(isSystem)
{
  const auto now = nowMillisUtc();
  createdAtMillis = now;
  updatedAtMillis = now;
}

void touchUpdated() { updatedAtMillis = nowMillisUtc(); }

template <typename Action>
void persist(Action& a) {
  dbo::field(a, key,         "key");
  dbo::field(a, displayName, "display_name");
```

```
    dbo::field(a, description, "description");
    dbo::field(a, system,      "system");

    dbo::field(a, createdAtMillis, "created_at_millis");
    dbo::field(a, updatedAtMillis, "updated_at_millis");

    // Many-to-many: users <-> roles
    dbo::hasMany(a, users, dbo::ManyToMany, "user_roles",
                 "role_id", "user_id");

    // Many-to-many: roles <-> permissions
    dbo::hasMany(a, permissions, dbo::ManyToMany, "role_permissions",
                 "role_id", "permission_id");
  }
};
```

13.2.5 Mapping registration

Map the class during application initialization:

```
#include <Wt/Dbo/Session.h>

static void mapEntities(dbo::Session& s) {
  // s.mapClass<User>("users");        // from previous section
  s.mapClass<Role>("roles");
  // s.mapClass<Permission>("permissions"); // next section
}
```

In production, apply the unique constraint and indices using migrations.

13.2.6 Role management rules (production policy)

Roles are security policy objects. Enforce the following rules in services/controllers:

Stable keys and immutability

- Treat `key` as immutable after creation.
- If you must rename a role, do it by updating display metadata, not the key.

System roles are protected

If `system == true`:

- prevent deletion,
- restrict permission changes to privileged administrators,
- require step-up authentication for modifications.

Role assignment auditing

When a user gains/loses roles:

- record an audit event (actor, target, role key, time),
- increment user's `sessionVersion` (or equivalent) to refresh authorization.

Role changes must invalidate permission caches

If permissions are cached in session:

- invalidate the cache,
- or revoke sessions using session version.

This ensures that role changes take effect immediately.

13.2.7 Practical modeling decisions

Avoid role explosion

Over-creating roles creates administrative complexity. Prefer:

- a small set of roles aligned with job functions,
- permissions for fine-grained differences,
- scoped roles only when multi-tenant boundaries require them.

Keep permissions separate

Do not encode permissions implicitly inside code branches like: `if(role == "admin")`. Instead:

- compute effective permissions from roles,
- check permissions for every privileged action.

This makes authorization logic scalable and testable.

13.2.8 Summary

The role model is the administrative unit of authorization. A production-grade `Role` entity must provide:

- a stable `key` for policy and migrations,
- display metadata and optional description,
- protection of built-in roles via `system`,
- many-to-many relations with users and permissions,

- operational support for auditing and session revocation.

With this role model in place, the next section can define the **Permission model** and the join structures that produce effective authorization decisions.

13.3 Relationships

Data modeling is not just defining tables; it is defining **relationships** that preserve business invariants, authorization boundaries, and long-term maintainability. In a User Management System (UMS), relationships are the structural core that connect:

- **identity** (User),
- **authorization** (Role, Permission),
- **session continuity** (SessionRecord / login sessions),
- **auditing** (AuditLog, security events),
- optional **tenancy boundaries** (Tenant/Org) in multi-tenant systems.

This section defines relationship types, schema patterns, constraints, indexing strategy, and complete `Wt::Dbo` relationship mapping for the live project.

13.3.1 Relationship design principles

1) Relationships must represent invariants

If the business rule says a user can have many roles, then the database must enforce it with:

- join tables,
- foreign keys,
- unique constraints to prevent duplicate associations.

2) Authorization relationships must be auditable

Role/permission changes affect security. Therefore:

- the relationship must be queryable efficiently,
- changes should be recorded (audit events),
- permission caches must be invalidated on updates.

3) Choose explicit join tables

Many-to-many relationships should use explicit join tables with meaningful names:

- stable schema,
- easier migrations,
- easier indexing and debugging,
- compatibility across database backends.

4) Control cascade behavior carefully

Cascades can be dangerous in security systems:

- do not cascade-delete users unless you have a strict data retention policy,
- prefer soft-delete or deactivation for identities,
- cascade delete may be acceptable for ephemeral session rows, but only if intended.

5) Model direction and navigation in the ORM

In an ORM, relationships affect:

- lazy loading and performance,
- query shape and N+1 risks,
- domain model clarity.

Always decide:

- which side "owns" the relationship,
- what navigation paths are required by use-cases.

13.3.2 Core relationships in the UMS

For the live project, we model the following core relationships:

User ↔ Role (many-to-many)

A user can have multiple roles, and a role can be assigned to many users.

Schema representation

- Table: `users`
- Table: `roles`
- Join table: `user_roles` with `user_id`, `role_id`

Constraints

- unique(`user_id`, `role_id`) to prevent duplicates,
- foreign keys referencing `users(id)` and `roles(id)`.

Indexes

- index on `user_id` (for loading roles by user),
- index on `role_id` (for listing users by role).

Role ↔ Permission (many-to-many)

Roles group permissions; permissions can belong to multiple roles.

Schema representation

- Table: `roles`
- Table: `permissions`
- Join table: `role_permissions` with `role_id`, `permission_id`

Constraints and indexes Same pattern as `user_roles`:

- unique(`role_id`, `permission_id`)
- index on each foreign key.

User → SessionRecord (one-to-many)

A user can have many active sessions (multiple devices). Each session belongs to one user.

Schema representation

- Table: `sessions`
- Foreign key: `sessions.user_id` references `users.id`

Constraints

- unique session identifier: `sessions.session_id` unique,
- index on `user_id` for listing sessions by user,
- index on `last_access_at` for cleanup/expiry jobs.

User → AuditLog (one-to-many, optional but recommended)

Audit rows should include:

- actor user id (who performed the action),
- target user id (which user was affected),
- event type, timestamp, and metadata.

This is optional in the live project if you keep audit in logs, but for production systems it is recommended as a database relation.

13.3.3 Relationship implementation in `Wt::Dbo`

In `Wt::Dbo`, relationships are declared inside `persist()` using:

- `dbo::belongsTo` for many-to-one ownership,
- `dbo::hasMany` for one-to-many and many-to-many collections.

Many-to-many: User and Role

User side

```
class Role;

class User {
public:
  dbo::collection< dbo::ptr<Role> > roles;

  template <typename Action>
  void persist(Action& a) {
    // other fields...

    dbo::hasMany(a, roles, dbo::ManyToMany, "user_roles",
                 "user_id", "role_id");
  }
};
```

Role side

```
class User;

class Role {
public:
  dbo::collection< dbo::ptr<User> > users;

  template <typename Action>
  void persist(Action& a) {
    // other fields...

    dbo::hasMany(a, users, dbo::ManyToMany, "user_roles",
                 "role_id", "user_id");
  }
```

```
};
```

Join table naming and key order Always keep join table naming and column order consistent across both sides. If they disagree, mapping becomes unreliable.

Many-to-many: Role and Permission

Role side

```
class Permission;

class Role {
public:
  dbo::collection< dbo::ptr<Permission> > permissions;

  template <typename Action>
  void persist(Action& a) {
    // other fields...

    dbo::hasMany(a, permissions, dbo::ManyToMany, "role_permissions",
                 "role_id", "permission_id");
  }
};
```

Permission side

```
class Role;

class Permission {
public:
  dbo::collection< dbo::ptr<Role> > roles;
```

```
  template <typename Action>
  void persist(Action& a) {
    // other fields...

    dbo::hasMany(a, roles, dbo::ManyToMany, "role_permissions",
                 "permission_id", "role_id");
  }
};
```

One-to-many: User and SessionRecord

This is typically modeled as:

- `SessionRecord` **belongs to** `User`,
- `User` **has many** `SessionRecord`.

SessionRecord side (many-to-one)

```
class User;

class SessionRecord {
public:
  std::string sessionId;
  std::int64_t loginAtMillis{0};
  std::int64_t lastAccessMillis{0};

  dbo::ptr<User> user;

  template <typename Action>
```

```
  void persist(Action& a) {
    dbo::field(a, sessionId, "session_id");
    dbo::field(a, loginAtMillis, "login_at_millis");
    dbo::field(a, lastAccessMillis, "last_access_millis");

    dbo::belongsTo(a, user, "user_id");
  }
};
```

User side (one-to-many)

```
class SessionRecord;

class User {
public:
  dbo::collection< dbo::ptr<SessionRecord> > sessions;

  template <typename Action>
  void persist(Action& a) {
    // other fields...
    dbo::hasMany(a, sessions, dbo::ManyToOne, "user_id");
  }
};
```

13.3.4 Relationship correctness constraints (database-level)

Even if your ORM can manage relationships, production correctness requires database constraints.

Join table uniqueness

Each join table should enforce uniqueness:

- `user_roles`: unique(`user_id`, `role_id`)
- `role_permissions`: unique(`role_id`, `permission_id`)

This prevents silent duplicates that can inflate permission sets and break auditing.

Foreign keys

Foreign keys enforce referential integrity:

- disallow join rows pointing to deleted or nonexistent entities,
- provide predictable behavior during migrations and cleanup.

Cascade policy guidelines

- **User deletion**: avoid cascading; prefer deactivation.
- **Role deletion**: restrict if role is in use; consider `system` roles undeletable.
- **Session deletion**: safe to cascade when a user is deactivated only if your retention policy allows.

13.3.5 Performance and loading patterns

Relationships influence performance. Practical guidelines:

Avoid N+1 in authorization

Authorization requires loading roles and permissions. Do not load them one by one:

- load effective permissions in one query (via join),
- cache permission keys in session,
- invalidate on role/permission changes.

Explicit loading inside transactions

Relations are often lazy-loaded. In critical paths:

- load relations explicitly inside a `dbo::Transaction`,
- build a view model,
- close the transaction before rendering large UI trees.

13.3.6 Relationship-driven authorization model

The effective authorization computation is:

$$\text{User Permissions} = \bigcup_{\text{role} \in \text{User Roles}} \text{Role Permissions}$$

This formula is the reason these relationships must remain consistent and indexed. The application relies on predictable joins.

13.3.7 Operational rules for relationship updates

When relationships change, you must enforce runtime consequences:

Role assigned/removed from user

- record audit event,
- increment user's `sessionVersion` (or revoke sessions),
- invalidate permission cache.

Permission assigned/removed from role

- record audit event,
- revoke sessions for all users in that role (or increment version of affected users),
- invalidate caches.

In large systems, this may require an asynchronous job; in smaller systems, it can be done synchronously.

13.3.8 Summary

Relationships in the UMS data model are the structure that makes authentication and authorization correct:

- **User** $\leftrightarrow$ **Role** (many-to-many) via `user_roles`,
- **Role** $\leftrightarrow$ **Permission** (many-to-many) via `role_permissions`,
- **User** $\rightarrow$ **SessionRecord** (one-to-many) via `sessions.user_id`,
- optional audit relationships for traceability.

A production-grade Wt::Dbo implementation requires:

- explicit ORM mapping with consistent join table definitions,
- database constraints (unique keys, foreign keys),
- indexes aligned with query patterns,
- deterministic cache invalidation and session revocation on relationship changes.

These relationship foundations enable the next steps: permission modeling, efficient permission queries, and production-ready migration strategies.

13.4 Lightweight migrations

A production database schema is a living artifact. It changes as features evolve, bugs are fixed, and security requirements grow. If schema evolution is handled ad-hoc (manual edits, `createTables()` reruns, or inconsistent scripts), the system eventually becomes un-deployable, non-reproducible, and dangerous to maintain.
This section presents a **lightweight migration system** suitable for Wt::Dbo-based applications:

- minimal moving parts,
- deterministic schema versioning,
- safe application startup behavior,
- compatible with multiple database backends,
- integrates naturally with a modern C++ codebase and CI/CD.

The goal is not to replace heavy migration frameworks, but to build an *auditable, repeatable, production-safe* mechanism that is easy to understand and maintain.

13.4.1 Why `createTables()` is not enough

`Wt::Dbo::Session::createTables()` is useful for:

- prototypes,
- quick local development,
- test environments where schema resets are acceptable.

It is not sufficient for production because:

- it does not provide a robust incremental history of schema changes,
- it does not naturally support data migrations (transforming existing rows),
- it is not designed for controlled rollbacks,
- it can diverge from a production schema over time.

Therefore, production systems should adopt versioned migrations even if they keep `createTables()` for development tests.

13.4.2 Definition: lightweight migrations

A lightweight migration system has these properties:

- **Versioned**: schema state is represented by an integer version.
- **Ordered**: migrations apply strictly in ascending order.
- **Idempotent or guarded**: each migration runs at most once.
- **Transactional when possible**: a migration either fully applies or does not apply.
- **Auditable**: applied migrations are recorded in a database table.
- **Deterministic**: deployments produce identical schema across environments.

13.4.3 Core pattern: migration ledger table

The simplest reliable approach is a single **migration ledger** table:

- records current schema version,
- records which migrations were applied,
- stores timestamps and optional description.

A common minimal schema:

- table `schema_migrations`
- columns:
  - `version` (integer, primary key),
  - `applied_at_millis` (integer),
  - `name` (string, optional),
  - `checksum` (string, optional, for integrity).

You can keep it even simpler (single-row table with current version), but recording each migration gives stronger auditability.

13.4.4 Migration execution strategy

When to run migrations

Run migrations:

- at application startup, before serving requests, or
- as a dedicated admin command/tool in your deployment pipeline.

For production-grade systems, prefer a deployment step (explicit migration command) to avoid applying schema changes during peak traffic. However, for small systems, safe startup migrations can be acceptable if:

- migrations are fast,
- migrations are guarded and transactional,
- only one instance runs migrations at a time (leader lock).

Single-writer rule

Never allow multiple application instances to run migrations concurrently. Ensure one of:

- a maintenance window with a single instance,
- a deployment pipeline step that runs migrations once,
- a database advisory lock (backend-dependent).

13.4.5 Lightweight migration framework in C++

We now define a minimal migration framework that:

- uses `dbo::Session` for connectivity,
- runs raw SQL for schema changes (portable and explicit),
- records each applied version,
- fails fast on error.

Migration interface

```
#pragma once
#include <cstdint>
#include <functional>
#include <string>
#include <vector>

struct Migration {
  std::int32_t version{0};
  std::string name;
  std::function<void()> up;   // apply migration
};
```

Migration runner contract

```
#pragma once
#include <Wt/Dbo/Dbo.h>
#include <Wt/Dbo/Transaction.h>

namespace dbo = Wt::Dbo;

class MigrationRunner {
public:
  explicit MigrationRunner(dbo::Session& session)
    : session_(session) {}

  void ensureLedgerTable();
  std::int32_t currentVersion();
  void applyAll(const std::vector<Migration>& migrations);
```

```
private:
  dbo::Session& session_;

  void recordApplied(std::int32_t version, const std::string& name);
  bool isApplied(std::int32_t version);
};
```

Ledger creation and version detection

Wt::Dbo provides query support. For portability across backends, use raw SQL cautiously. Below is a backend-agnostic approach that relies on:

- creating the ledger if missing,
- reading max version from the ledger.

```
#include <Wt/Dbo/Session.h>
#include <Wt/Dbo/Transaction.h>

namespace dbo = Wt::Dbo;

void MigrationRunner::ensureLedgerTable() {
  dbo::Transaction t(session_);

  // Minimal SQL; adjust types per backend if needed.
  // Many backends accept INTEGER and VARCHAR.
  session_.execute(
    "CREATE TABLE IF NOT EXISTS schema_migrations ("
    "version INTEGER PRIMARY KEY, "
    "applied_at_millis BIGINT NOT NULL, "
```

```
    "name VARCHAR(200) NOT NULL"
    ")"
  );

  t.commit();
}

std::int32_t MigrationRunner::currentVersion() {
  dbo::Transaction t(session_);

  // If no rows exist, version is 0.
  int v = 0;
  auto q =
  ↪  session_.query<int>("SELECT COALESCE(MAX(version), 0) FROM schema_migrations");
  v = q.resultValue();

  t.commit();
  return static_cast<std::int32_t>(v);
}

bool MigrationRunner::isApplied(std::int32_t version) {
  dbo::Transaction t(session_);
  auto q =
  ↪  session_.query<int>("SELECT COUNT(1) FROM schema_migrations WHERE version = ?")
           .bind(version);
  const int count = q.resultValue();
  t.commit();
  return count != 0;
}
```

```
void MigrationRunner::recordApplied(std::int32_t version, const std::string& name)
↪  {
  const auto now = static_cast<long
  ↪  long>(std::chrono::duration_cast<std::chrono::milliseconds>(
                    std::chrono::system_clock::now().time_since_epoch()).count());


  ↪  session_.execute("INSERT INTO schema_migrations(version, applied_at_millis, name) VA
    .bind(version)
    .bind(now)
    .bind(name);
}
```

Applying migrations safely

```
#include <algorithm>
#include <stdexcept>

void MigrationRunner::applyAll(const std::vector<Migration>& migrations) {
  ensureLedgerTable();

  // Make a sorted copy to guarantee order.
  std::vector<Migration> ms = migrations;
  std::sort(ms.begin(), ms.end(),
            [](const Migration& a, const Migration& b) { return a.version <
            ↪  b.version; });

  // Ensure versions are unique and positive.
  for (std::size_t i = 0; i < ms.size(); ++i) {
```

```
    if (ms[i].version <= 0) {
      throw std::runtime_error("Migration version must be positive");
    }
    if (i > 0 && ms[i].version == ms[i - 1].version) {
      throw std::runtime_error("Duplicate migration version");
    }
  }

  for (const auto& m : ms) {
    if (isApplied(m.version)) {
      continue; // already applied
    }

    // Apply each migration in a transaction where possible.
    dbo::Transaction t(session_);
    try {
      m.up();                 // run schema changes / data transforms
      recordApplied(m.version, m.name);
      t.commit();
    } catch (...) {
      // rollback on scope exit
      throw;
    }
  }
}
```

13.4.6 Writing migrations: schema + data

A lightweight system must support both:

Schema migrations

Examples:

- create tables,
- add columns,
- create indexes and unique constraints,
- add foreign keys.

Data migrations

Examples:

- backfill `session_version` default values,
- canonicalize emails,
- migrate role keys.

13.4.7 Example migrations for the UMS

Below is a minimal migration set consistent with the entities from Chapter 13.

```
#include <vector>
#include <Wt/Dbo/Session.h>

static std::vector<Migration> buildMigrations(dbo::Session& s) {
  std::vector<Migration> ms;

  ms.push_back(Migration{
    1,
```

```
  "create users table",
  [&] {
    s.execute(
      "CREATE TABLE IF NOT EXISTS users ("
      "id INTEGER PRIMARY KEY, "
      "email VARCHAR(320) NOT NULL, "
      "display_name VARCHAR(200) NOT NULL, "
      "password_hash VARCHAR(300) NOT NULL, "
      "active BOOLEAN NOT NULL, "
      "locked BOOLEAN NOT NULL, "
      "failed_login_count INTEGER NOT NULL, "
      "lock_until_millis BIGINT NULL, "
      "session_version BIGINT NOT NULL, "
      "created_at_millis BIGINT NOT NULL, "
      "updated_at_millis BIGINT NOT NULL, "
      "last_login_at_millis BIGINT NULL, "
      "locale VARCHAR(16) NOT NULL, "
      "time_zone VARCHAR(64) NOT NULL"
      ")"
    );

   s.execute("CREATE UNIQUE INDEX IF NOT EXISTS ux_users_email ON users(email)");
  }
});

ms.push_back(Migration{
  2,
  "create roles and permissions",
  [&] {
```

```
    s.execute(
      "CREATE TABLE IF NOT EXISTS roles ("
      "id INTEGER PRIMARY KEY, "
      "key VARCHAR(120) NOT NULL, "
      "display_name VARCHAR(200) NOT NULL, "
      "description VARCHAR(500) NOT NULL, "
      "system BOOLEAN NOT NULL, "
      "created_at_millis BIGINT NOT NULL, "
      "updated_at_millis BIGINT NOT NULL"
      ")"
    );

    s.execute("CREATE UNIQUE INDEX IF NOT EXISTS ux_roles_key ON roles(key)");
  }
});

ms.push_back(Migration{
  3,
  "create user_roles join table",
  [&] {
    s.execute(
      "CREATE TABLE IF NOT EXISTS user_roles ("
      "user_id INTEGER NOT NULL, "
      "role_id INTEGER NOT NULL"
      ")"
    );


    ↪  s.execute("CREATE UNIQUE INDEX IF NOT EXISTS ux_user_roles ON user_roles(user_id
```

```
      ↪ s.execute("CREATE INDEX IF NOT EXISTS ix_user_roles_user ON user_roles(user_id)")
      ↪ s.execute("CREATE INDEX IF NOT EXISTS ix_user_roles_role ON user_roles(role_id)")
    }
  });

  return ms;
}
```

13.4.8 Migration safety rules

Rule 1: backward-compatible changes

Prefer additive changes:

- add nullable columns first,
- backfill data,
- then enforce `NOT NULL` or constraints.

Rule 2: avoid long locks

Schema operations can lock tables. Mitigate by:

- scheduling heavy migrations in maintenance windows,
- creating indexes carefully (backend-dependent),
- splitting large migrations into smaller steps.

Rule 3: treat migrations as code

- version-control all migrations,
- run them in CI on a fresh database,
- validate schema version equals expected version after deploy.

Rule 4: define a rollback plan

Lightweight migrations often omit automatic down migrations. Still, you must plan:

- backups before applying destructive changes,
- feature flags to tolerate partial rollout,
- explicit down scripts for critical releases if required.

Rule 5: fail fast and stop the deployment

If any migration fails:

- stop applying subsequent migrations,
- keep the database in the pre-migration state through rollback,
- surface error to operators.

13.4.9 Recommended integration patterns

Pattern A: Dedicated migration command

Create a small CLI tool:

- opens DB connection,

- maps entities if needed,
- runs `MigrationRunner::applyAll()`.

Then call it from CI/CD before starting application instances.

Pattern B: Startup migrations with a lock

If running at startup:

- acquire a database lock or ensure single instance,
- apply migrations,
- start serving requests.

13.4.10 Summary

Lightweight migrations provide a production-safe schema evolution mechanism without heavy tooling:

- A `schema_migrations` ledger records applied versions.
- Migrations are ordered, transactional when possible, and applied exactly once.
- Schema and data migrations are both supported.
- Concurrency is controlled by ensuring a single migration runner.
- Production safety requires backward-compatible steps, lock awareness, and CI validation.

With this migration framework in place, the UMS data layer can evolve confidently as roles, permissions, sessions, and audit features grow in the next chapters.

A Clean Database Layer

14.1 Database singleton or connection pool

Designing the database layer for a production server-side application is a critical architectural choice that directly affects scalability, performance, reliability, and maintainability. In Wt-based applications using `Wt::Dbo` or any database access layer, one of the earliest decisions is:

> **Should the application use a database connection singleton or a connection pool?**

This section analyzes both patterns in depth, explains their characteristics, and demonstrates how to implement them safely in a Modern C++ Web Development context with Wt.

14.1.1 Connection basics and web application demands

Web applications, especially those designed for many concurrent users, have the following database demands:

- **Multiple concurrent requests** each needing database access.
- **Short-lived transactions** with minimal contention.

- **Efficient connection reuse** to avoid high connection setup costs.
- **Scalability**: the ability to handle increasing throughput with reasonable resource consumption.

Treat each incoming HTTP request (or internal Wt event) as a potential database client. Under load, a single database connection (singleton) quickly becomes a bottleneck, whereas connection pooling allows many clients to share a pool of ready-to-use connections.

14.1.2 Singleton connection pattern

The **database singleton** pattern uses a single shared connection object for the lifetime of the application.

Characteristics

- **Single connection**: only one underlying database session/connection exists.
- **Global access**: typically accessed via a static or globally shared object.
- **Simplest implementation**: no pooling logic, no resource management beyond the one connection.

Limitations

- **Concurrency bottleneck**: only one command can run at a time if the backend connection is not thread-safe.
- **Thread-safety concerns**: many database drivers are not thread-safe on a single connection without explicit serialization.
- **High latency under load**: requests must wait for others to complete their database interactions.

- **Poor scalability**: unfit for real production workflows with many simultaneous users.

Use cases

Singleton connections can be acceptable for:

- low-throughput internal tools,
- administrative scripts,
- development prototypes where simplicity is paramount.

They should not be the default for production services.

14.1.3 Connection pool pattern

A **connection pool** maintains a set of database connections that can be reused by concurrent requests. Each worker borrows a connection, executes its work, and returns the connection to the pool.

Characteristics

- **Multiple connections**: allows concurrent DB usage.
- **Resource reuse**: reduces overhead of connection setup/teardown.
- **Thread-aware**: works safely in multi-threaded environments.
- **Configurable sizing**: pool size can be tuned based on expected load and backend limits.

Pooling strategies

- **Fixed-size pool**: create a fixed number of connections at startup.
- **Dynamic-size pool**: grow and shrink based on demand (within configured limits).
- **Timeouts**: borrow operations time out to avoid stalling when the pool is exhausted.

Advantages for web applications

- **Concurrency**: many requests can proceed in parallel, limited only by pool size and backend capacity.
- **Performance**: reuse reduces connection latency.
- **Stability**: controlled resource usage prevents overload.
- **Thread safety**: each borrowed connection is used by only one worker at a time.

When not to pool

Pooling may be excessive for:

- CLI tools with a single operation,
- unit tests with isolated databases,
- very low throughput services where overhead outweighs benefits.

Even then, a simple pool with one connection behaves effectively like a singleton but with unified semantics.

14.1.4 Implementing a connection pool in Modern C++

In a Wt application, the database layer is typically behind `dbo::Session`. Each session holds one backend connection. Consequently, pooling should manage multiple sessions.

Core API design

Design a pool that:

- constructs backend connections on demand,
- hands out sessions safely to request handlers,
- returns sessions to the pool on release,
- optionally validates liveliness.

```
#pragma once
#include <condition_variable>
#include <memory>
#include <mutex>
#include <queue>
#include <string>
#include <stdexcept>

template <typename SessionFactory>
class DbConnectionPool {
public:
  DbConnectionPool(SessionFactory factory,
                   std::size_t maxSize)
    : factory_(factory),
      maxSize_(maxSize) {}
```

```
std::unique_ptr<typename SessionFactory::SessionType> acquire() {
  std::unique_lock lock(mutex_);

  if (!pool_.empty()) {
    auto session = std::move(pool_.front());
    pool_.pop();
    return session;
  }

  if (totalCreated_ < maxSize_) {
    auto session = factory_();
    ++totalCreated_;
    return session;
  }

  if (!cond_.wait_for(lock, timeout_,
                      [&] { return !pool_.empty(); })) {
    throw std::runtime_error("DB pool timeout");
  }

  auto session = std::move(pool_.front());
  pool_.pop();
  return session;
}

void release(std::unique_ptr<typename SessionFactory::SessionType> session) {
  std::lock_guard lock(mutex_);
  pool_.push(std::move(session));
```

```
    cond_.notify_one();
  }

  void setTimeout(std::chrono::milliseconds t) { timeout_ = t; }

private:
  SessionFactory factory_;
  std::size_t maxSize_;
  std::size_t totalCreated_{0};

  std::chrono::milliseconds timeout_{std::chrono::milliseconds(5000)};

  std::mutex mutex_;
  std::condition_variable cond_;
  std::queue<std::unique_ptr<typename SessionFactory::SessionType>> pool_;
};
```

Here, `SessionFactory` is a functor that returns a new session (with connection) wrapped in a unique pointer. `SessionType` is a typedef within the factory for the session class.

Usage pattern

```
struct MySessionFactory {
  using SessionType = Wt::Dbo::Session;

  MySessionFactory(const std::string& connInfo)
    : connInfo_(connInfo) {}

  std::unique_ptr<SessionType> operator()() const {
    auto session = std::make_unique<SessionType>();
```

```
    auto backend = std::make_unique<Wt::Dbo::backend::Postgres>(connInfo_);
    backend->setProperty("show-queries", "false");
    session->setConnection(std::move(backend));
    mapEntities(*session); // register mappings here
    return session;
  }

private:
  std::string connInfo_;
};

static DbConnectionPool<MySessionFactory> makePool(const std::string& connInfo) {
  return DbConnectionPool<MySessionFactory>(MySessionFactory(connInfo), 20);
}
```

Application code can then:

```
auto session = pool.acquire();  // borrow
{
  Wt::Dbo::Transaction t(*session);
  // do DB work
  t.commit();
}
pool.release(std::move(session)); // return
```

This pattern ensures each worker gets a separate session and transaction context.

14.1.5 Integration with Wt application sessions

In Wt, request handlers and internal events are served by threads that may need database access. The recommended pattern is:

- each request borrows a DB session from the pool,
- performs transactions in the scope of that session,
- returns the session at the end of the request/event handler.

This avoids sharing a session object across concurrent requests.

14.1.6 Error and retry policies

A robust pool should handle:

- backend disconnections (detect and recreate),
- transient errors (retry logic),
- max pool exhaustion (failfast or backpressure).

For example, if a session becomes invalid, the pool can drop it and create a new one:

```
void release(std::unique_ptr<typename SessionFactory::SessionType> session) {
  if (!session->connectionIsHealthy()) {
    --totalCreated_;
    // do not re-add to pool
    cond_.notify_one();
    return;
  }
  std::lock_guard lock(mutex_);
  pool_.push(std::move(session));
  cond_.notify_one();
}
```

The hypothetical `connectionIsHealthy()` checks driver-specific state.

14.1.7 Singleton vs pool: summary comparison

Feature	Singleton	Connection Pool
Concurrency support	poor	strong
Resource reuse	limited	excellent
Thread safety	complex	easier (borrow pattern)
Performance under load	bottleneck	scalable
Implementation complexity	minimal	moderate
Production readiness	no	yes

14.1.8 Advanced patterns: scoped sessions

For large applications, further decouple:

- **Repository-layer sessions**: inject sessions into repository objects per request.
- **Per-transaction sessions**: borrow at unit-of-work boundaries only.
- **Async job workers**: use separate pools or dedicated connections.

14.1.9 Pooling in distributed environments

In multi-instance deployments behind load balancers:

- each instance has its own pool, sized according to CPU and expected concurrency,
- backend database must support required connection count,
- shared sessions across instances require external session store (Redis, database) if you avoid sticky sessions.

14.1.10 Testing and mocks

For tests:

- use an in-memory SQLite pool with minimal connections,
- override factory to provide test doubles,
- wrap transactions in rollback for isolation.

```
struct TestSessionFactory {
  using SessionType = Wt::Dbo::Session;

  std::unique_ptr<SessionType> operator()() const {
    auto session = std::make_unique<SessionType>();
    auto backend = std::make_unique<Wt::Dbo::backend::Sqlite3>(":memory:");
    session->setConnection(std::move(backend));
    mapEntities(*session);
    dbo::Transaction t(*session);
    session->createTables();
    t.commit();
    return session;
  }
};
```

14.1.11 Summary

Choosing between a database singleton and a connection pool determines how your Wt application scales:

- **Singleton** is simple but unsuitable for production.

- **Connection pool** provides concurrency, performance, and thread safety.

A connection pool pattern:

- borrows sessions per request/job,
- reduces connection overhead,
- integrates naturally with `Wt::Dbo::Session` and transactions,
- enables robust error handling and retry policies.

In a clean database layer, pooling is essential for production-grade scalability, observability, and maintainability.

14.2 Separating SQL from business logic

A production-grade application must treat database access as an **infrastructure concern**, not a business concern. When SQL is interleaved with business rules inside controllers and UI code, the system becomes:

- hard to test (logic depends on the database),
- hard to refactor (SQL fragments spread everywhere),
- fragile under change (schema evolution breaks many unrelated locations),
- vulnerable to subtle security bugs (authorization checks drift from data access),
- performance-unpredictable (queries appear inside loops and UI handlers).

This section defines a clean separation strategy tailored to Modern C++ and Wt applications using `Wt::Dbo`:

- a layered architecture,
- repositories and domain services,
- query encapsulation patterns,
- transaction boundary rules,
- testable abstractions and error mapping.

14.2.1 The architectural goal

We want:

- Business logic to be expressed in terms of **domain concepts** (User, Role, Session, Policy).
- SQL and ORM details to be isolated behind **data access interfaces**.
- Controllers to orchestrate use-cases, not write queries.
- UI to render results, not decide persistence rules.

A clean database layer yields:

- deterministic transaction scopes,
- consistent error handling,
- reuse of query logic across use-cases,
- safer authorization enforcement.

14.2.2 Layered model: the minimal clean separation

A practical architecture for a Wt application is:

- **UI layer**: Wt widgets/pages (views).
- **Controller layer**: request orchestration, navigation, mapping domain errors to UI.
- **Domain services**: business rules and invariants (lockout policy, role assignment policy).
- **Repositories**: data access operations (load user, list roles, persist updates).
- **Infrastructure**: `dbo::Session`, connection pool, migrations, logging, crypto.

14.2.3 A common anti-pattern

Example of what *not* to do: SQL or Dbo calls inside UI code.

```
// Anti-pattern: UI code directly performing data access and policy
loginButton->clicked().connect([=] {
  dbo::Transaction t(session);
  auto u =
  ↪  session.find<User>().where("email = ?").bind(emailEdit->text().toUTF8());

  if (!u) {
    showError("Invalid credentials");
    return;
  }

  // business policy in UI
  if (!u->active || u->locked) {
    showError("Account disabled");
```

```
    return;
  }

  // more SQL and logic...
  t.commit();
});
```

Problems:

- cannot unit-test without DB,
- transaction scope is tied to UI event handler lifetime,
- policy is duplicated and scattered,
- query shape becomes unpredictable.

14.2.4 Clean separation pattern

We split responsibilities:

- **Repository**: provides *data retrieval and persistence* as functions.
- **Domain service**: provides *business rules* using repositories.
- **Controller**: orchestrates use-case flow and maps results to UI.

14.2.5 Repository design

A repository:

- hides SQL details,
- returns domain-friendly objects (snapshots, ids, view models),

- never contains UI code,
- does not decide policy (unless purely data integrity).

Repository interface

```
#pragma once
#include <cstdint>
#include <optional>
#include <string>
#include <vector>

struct UserSecuritySnapshot {
  std::int64_t id{0};
  std::string email;
  std::string passwordHash;
  bool active{false};
  bool locked{false};
  std::int32_t failedLoginCount{0};
  std::optional<std::int64_t> lockUntilMillis;
  std::int64_t sessionVersion{0};
};

class IUserRepository {
public:
  virtual ~IUserRepository() = default;

  virtual std::optional<UserSecuritySnapshot>
  findForLoginByEmail(const std::string& canonicalEmail) = 0;
```

```
  virtual void updateLoginSuccess(std::int64_t userId, std::int64_t nowMillis) = 0;
  virtual void updateLoginFailure(std::int64_t userId, std::int64_t nowMillis,
                                  std::int32_t newFailCount,
                                  std::optional<std::int64_t> lockUntilMillis,
                                  bool locked) = 0;

  virtual void incrementSessionVersion(std::int64_t userId) = 0;
};
```

Notice:

- no SQL appears in higher layers,
- returned type is a snapshot, not an ORM pointer,
- update functions represent domain actions in data terms.

Wt::Dbo implementation of repository

```
#pragma once
#include <Wt/Dbo/Dbo.h>
#include <Wt/Dbo/Transaction.h>

namespace dbo = Wt::Dbo;

class DboUserRepository final : public IUserRepository {
public:
  explicit DboUserRepository(dbo::Session& session) : session_(session) {}

  std::optional<UserSecuritySnapshot>
  findForLoginByEmail(const std::string& canonicalEmail) override {
```

```
  dbo::Transaction t(session_);

  dbo::ptr<User> u = session_.find<User>()
    .where("email = ?").bind(canonicalEmail);

  if (!u) {
    return std::nullopt;
  }

  UserSecuritySnapshot s;
  s.id = u.id();
  s.email = u->email;
  s.passwordHash = u->passwordHash;
  s.active = u->active;
  s.locked = u->locked;
  s.failedLoginCount = u->failedLoginCount;
  s.lockUntilMillis = u->lockUntilMillis;
  s.sessionVersion = u->sessionVersion;

  t.commit();
  return s;
}

void updateLoginSuccess(std::int64_t userId, std::int64_t nowMillis) override {
  dbo::Transaction t(session_);

  dbo::ptr<User> u = session_.load<User>(userId);
  u.modify()->failedLoginCount = 0;
  u.modify()->locked = false;
```

```
  u.modify()->lockUntilMillis.reset();
  u.modify()->lastLoginAtMillis = nowMillis;
  u.modify()->updatedAtMillis = nowMillis;

  t.commit();
}

void updateLoginFailure(std::int64_t userId, std::int64_t nowMillis,
                        std::int32_t newFailCount,
                        std::optional<std::int64_t> lockUntilMillis,
                        bool locked) override {
  dbo::Transaction t(session_);

  dbo::ptr<User> u = session_.load<User>(userId);
  u.modify()->failedLoginCount = newFailCount;
  u.modify()->locked = locked;
  u.modify()->lockUntilMillis = lockUntilMillis;
  u.modify()->updatedAtMillis = nowMillis;

  t.commit();
}

void incrementSessionVersion(std::int64_t userId) override {
  dbo::Transaction t(session_);
  dbo::ptr<User> u = session_.load<User>(userId);
  u.modify()->sessionVersion += 1;
  u.modify()->updatedAtMillis = nowMillisUtc();
  t.commit();
}
```

```
private:
  dbo::Session& session_;
};
```

This implementation contains all ORM/SQL knowledge, while the rest of the app depends only on `IUserRepository`.

14.2.6 Domain service design

A domain service:

- defines policies (lockout, session revocation),
- consumes repository interfaces,
- returns domain outcomes (success, invalid credentials, locked, disabled),
- does not know about Wt UI.

Login policy service example

```
#pragma once
#include <cstdint>
#include <optional>
#include <string>

enum class LoginResult {
  Success,
  InvalidCredentials,
  Locked,
  Disabled
```

```
};

struct LoginOutcome {
  LoginResult result{LoginResult::InvalidCredentials};
  std::optional<std::int64_t> userId;
  std::int64_t sessionVersion{0};
};

class PasswordVerifier {
public:
  virtual ~PasswordVerifier() = default;
  virtual bool verify(const std::string& password, const std::string& hash) = 0;
};

class LoginService {
public:
  LoginService(IUserRepository& repo, PasswordVerifier& verifier)
    : repo_(repo), verifier_(verifier) {}

  LoginOutcome login(const std::string& canonicalEmail,
                     const std::string& password,
                     std::int64_t nowMillis)
  {
    auto s = repo_.findForLoginByEmail(canonicalEmail);
    if (!s) return {LoginResult::InvalidCredentials, std::nullopt, 0};

    if (!s->active) return {LoginResult::Disabled, std::nullopt, 0};

    if (s->locked) {
```

```
      const bool lockActive = !s->lockUntilMillis.has_value() || (nowMillis <
      ↪  *s->lockUntilMillis);
      if (lockActive) return {LoginResult::Locked, std::nullopt, 0};
    }

    if (!verifier_.verify(password, s->passwordHash)) {
      const auto newCount = s->failedLoginCount + 1;

      // Example policy: lock after 5 failures for 10 minutes (tune as needed).
      const bool doLock = (newCount >= 5);
      std::optional<std::int64_t> until = doLock ?
      ↪  std::optional<std::int64_t>(nowMillis + 10LL * 60LL * 1000LL)
                                                 : std::nullopt;

      repo_.updateLoginFailure(s->id, nowMillis, newCount, until, doLock);
      return {LoginResult::InvalidCredentials, std::nullopt, 0};
    }

    repo_.updateLoginSuccess(s->id, nowMillis);
    return {LoginResult::Success, s->id, s->sessionVersion};
  }

private:
  IUserRepository& repo_;
  PasswordVerifier& verifier_;
};
```

Now the login policy is unit-testable with a fake repository and a fake verifier.

14.2.7 Controller orchestration

The controller:

- uses domain services to execute the use-case,
- sets session state (logged-in user id, session version snapshot),
- maps errors to UI messages without leaking internal details.

```
class LoginController {
public:
  LoginController(LoginService& service, SessionState& sessionState)
    : service_(service), sessionState_(sessionState) {}

  bool handleLogin(const std::string& email,
                   const std::string& password,
                   std::int64_t nowMillis)
  {
    const auto canonicalEmail = canonicalizeEmail(email);
    const auto outcome = service_.login(canonicalEmail, password, nowMillis);

    if (outcome.result == LoginResult::Success) {
      sessionState_.onLogin(*outcome.userId, outcome.sessionVersion, nowMillis,
      ↪  makeSessionIdHint());
      return true;
    }

    // UI will decide what message to show:
    return false;
  }
```

```
private:
  LoginService& service_;
  SessionState& sessionState_;

  static std::string canonicalizeEmail(const std::string& s) {
    // Minimal example: trim and lowercase in your actual implementation.
    return s;
  }

  static std::string makeSessionIdHint() { return "sid"; }
};
```

No SQL appears here; only orchestration.

14.2.8 Transaction boundaries

One of the most important reasons to separate SQL from business logic is the ability to define consistent transaction boundaries.

Rule: transaction per use-case

- A use-case (e.g., *create user*, *assign role*, *disable account*) should execute under one outer transaction.
- Repository methods should assume they operate within that transaction, or they should be explicitly transactional by design.

For Wt::Dbo, you must choose one style:

Style A: outer transaction in controller/service

- Controller creates `dbo::Transaction`.
- Repositories do not create transactions internally.

Advantage: one transaction per use-case, simpler consistency. **Risk**: repository cannot be called outside a transaction.

Style B: transactional repository methods

- Each repository method starts its own transaction.

Advantage: simple usage, fewer prerequisites. **Risk**: multi-step use-cases cannot be atomic across multiple repo calls.
For production UMS, Style A is generally preferred for write use-cases. Style B can still be acceptable for simple reads.

14.2.9 Encapsulating query shapes

A clean data layer should treat queries as assets. Use these patterns:

Pattern 1: repository methods named after use-cases

Examples:

- `findForLoginByEmail()`,
- `loadUserWithRoles()`,
- `listUsersPaged()`,
- `loadEffectivePermissions(userId)`.

Pattern 2: query objects for complex queries

A query object isolates complex joins and paging.

```
struct UserListItem {
  std::int64_t id{0};
  std::string email;
  std::string displayName;
  bool active{false};
};

class IUserQuery {
public:
  virtual ~IUserQuery() = default;
  virtual std::vector<UserListItem> listPaged(int offset, int limit) = 0;
};
```

14.2.10 Error mapping and domain-safe failures

SQL/ORM errors should not leak to the UI. Enforce:

- repositories translate low-level errors into typed domain errors when appropriate,
- controllers log internal details and show generic user-safe messages.

Example domain error type:

```
enum class DataError {
  UniqueViolation,
  ForeignKeyViolation,
  NotFound,
  TransientFailure,
```

```
  FatalFailure
};
```

14.2.11 Testing benefits

When SQL is separated:

- domain services can be unit-tested without a database,
- repositories can be tested with integration tests (SQLite or container DB),
- controllers can be tested with mocks for domain services.

This gives a clean testing pyramid:

- many fast unit tests,
- fewer integration tests,
- minimal end-to-end tests.

14.2.12 Summary

Separating SQL from business logic is essential for a clean database layer in a Wt application:

- keep ORM/SQL in repositories,
- keep business rules in domain services,
- keep orchestration in controllers,
- keep UI purely for rendering and interaction,
- define transaction boundaries explicitly (prefer one transaction per use-case),

- encapsulate query shapes and map errors to domain-safe results.

This architecture makes the system more secure, more testable, easier to migrate, and more maintainable as the live project grows.

14.3 Error handling strategies

Error handling in the database layer is a core quality attribute of a production server-side application. In systems where persistence interacts with business logic and external clients (e.g., via web UI or APIs), database errors are not exceptions to ignore; they are operational events that must be classified, communicated, and acted upon correctly.
This section defines robust, maintainable, and production-grade error handling strategies for Modern C++ applications using Wt::Dbo or similar ORM/data layers.

14.3.1 Goals of error handling

A solid error handling strategy should satisfy:

- **Correct classification**: distinguish between transient, domain, constraint, and fatal errors.
- **Safe propagation**: do not expose internal error details to UI or API clients.
- **Actionability**: allow calling layers (controllers/services) to decide on retries, user messages, or rollback behavior.
- **Observability**: log sufficient context for debugging while avoiding leakage of sensitive data.
- **Consistency**: error responses should be structured and predictable across the application.

14.3.2 Error categories

Errors encountered in database operations typically fall into these categories:

1. Constraint violations

These occur when a database constraint is violated, such as:

- unique key violation,
- foreign key violation,
- check constraints on data ranges.

These are **domain-level errors**: they signal that the client attempted an operation that conflicts with schema or business invariants.

2. Transient errors

These include:

- connection timeouts,
- deadlocks,
- lock timeouts,
- temporary network blips.

Transient errors may succeed when retried under a backoff policy.

3. System errors

These are serious failures such as:

- database process crash,
- configuration errors (missing tables, columns),
- out-of-memory on the database server,
- ORM mapping inconsistency.

These should cause controlled failure of the operation and often stop further processing.

4. Not-found or concurrency misreads

When a referenced row no longer exists (deleted concurrently), some operations fail with a not-found-like error. This is semantically distinct from constraint violation: the row is absent rather than invalid.

14.3.3 Exception translation and domain error types

Low-level exceptions thrown by database drivers or Wt::Dbo should be translated into **domain error types** before they reach business logic. A domain error type is an enum or class that represents specific conditions.

```
enum class DbError {
  None,
  UniqueViolation,
  ForeignKeyViolation,
  NotFound,
  Deadlock,
  Timeout,
```

```
  Transient,
  Fatal
};
```

Repository layers should catch raw exceptions and map them:

```
DbError mapException(const std::exception& ex) {
  const std::string msg = ex.what();

  if (msg.find("unique constraint") != std::string::npos) {
    return DbError::UniqueViolation;
  }
  if (msg.find("foreign key") != std::string::npos) {
    return DbError::ForeignKeyViolation;
  }
  if (msg.find("deadlock") != std::string::npos) {
    return DbError::Deadlock;
  }
  if (msg.find("timeout") != std::string::npos) {
    return DbError::Timeout;
  }
  // Fallback
  return DbError::Fatal;
}
```

(Adapt the mapping to your backend error message conventions or use error codes when available.)

14.3.4 Repository-level strategies

Repositories should:

- encapsulate exception-to-error translation,
- return **expected/optional/error** result types instead of throwing for domain-level failures,
- propagate system failures for global handlers to log and abort use-cases.

Example return type:

```
template <typename T>
struct Result {
  std::optional<T> value;
  DbError error{DbError::None};
};
```

Usage in a repository method:

```
Result<UserSecuritySnapshot>
DboUserRepository::findForLoginByEmail(const std::string& email) {
  try {
    dbo::Transaction t(session_);
    // ORM find logic...
    t.commit();
    return {snapshot, DbError::None};
  } catch (const std::exception& ex) {
    return {std::nullopt, mapException(ex)};
  }
}
```

This pattern separates database exception handling from domain logic.

14.3.5 Service and controller response mapping

Domain services should interpret committed repository results and decide:

- whether to retry (transient errors),
- whether to present a domain error to the user (constraint violation),
- whether to escalate system errors.

Service example:

```
LoginOutcome LoginService::login(...) {
  auto res = repo_.findForLoginByEmail(canonicalEmail);
  if (res.error == DbError::UniqueViolation) {
    return {LoginResult::InvalidCredentials};
  }
  if (res.error == DbError::Timeout || res.error == DbError::Deadlock) {
    // transient: recommend retry
    throw TransientFailureException{};
  }
  if (res.error == DbError::Fatal) {
    throw SystemFailureException{};
  }
  // continue domain logic
}
```

Controllers catch transient and system exceptions, map to generic UI responses, and log details.

14.3.6 Transaction boundary and exception safety

Define clear transaction boundaries. Two models exist:

Model A: outer transaction per use-case

```
dbo::Transaction t(session_);
try {
  service_.doSomething(...);
  t.commit();
} catch (...) {
  // transaction rollbacks automatically on scope exit
  throw;
}
```

This ensures atomicity of the entire use-case.

Model B: repository internal transactions

Repositories start and commit transactions individually. This can simplify repository code but makes multi-step atomicity harder.
Prefer Model A for write-heavy use-cases.

14.3.7 Retry policies for transient errors

Transient errors often succeed on retry with a backoff. Implement an idempotent retry wrapper:

```
template <typename Fn>
auto retryOnTransient(Fn&& fn, int maxRetries = 3) {
  int attempt = 0;
  while (true) {
    try {
      return fn();
    } catch (const TransientFailureException&) {
      if (++attempt >= maxRetries) throw;
```

```
      std::this_thread::sleep_for(std::chrono::milliseconds(50 * attempt));
    }
  }
}
```

Wrap only idempotent operations to avoid unintended side effects.

14.3.8 Logging and observability

Errors should be logged with context:

- operation name,
- parameters (with sensitive data redacted),
- user identity/context if available,
- error classification (domain vs system vs transient),
- stack trace if available.

Example logging facade use:

```
try {
  repo_.updateSomething(...);
} catch (const std::exception& ex) {
  logger.error("updateSomething failed, user={}, reason={}",
               ctx.userId(), ex.what());
  throw;
}
```

Avoid logging raw passwords, tokens, or PII.

14.3.9 UI-safe error messages

UI or API clients should never see internal error text or stack traces. Instead, surface:

- Unable to complete request right now. (for system failures),
- Invalid input or state. (for domain/constraint violations),
- Please try again later. (for transient issues).

Keep responses consistent to avoid leaking internal structure.

14.3.10 Mapping backend-specific errors

When backends support error codes (SQLSTATE, driver-specific), use structured codes rather than string matching. Abstract them in mapping functions to `DbError`.

14.3.11 Testing error handling

Cover error paths with both unit and integration tests:

- simulate constraint violations (duplicate email),
- simulate transient errors via mocks,
- simulate system errors by disconnecting database,
- assert that service/controller map errors correctly.

Mocks help validate without a database; integration tests validate real ORM behavior.

14.3.12 Summary

A production-grade database error handling strategy involves:

- classifying errors into domain, transient, and system levels,
- translating low-level exceptions into domain error types,
- encapsulating SQL/ORM errors in repository results,
- retrying transient failures where appropriate,
- logging with context and redacting sensitive data,
- surfacing safe, consistent messages to clients,
- defining clear transaction boundaries,
- testing both success and error paths extensively.

When applied consistently, these strategies make the database layer robust, observable, maintainable, and safe in production deployments.

Part VII

Business Logic Layer

Service Layer Design

15.1 Why a service layer matters

In production-grade server-side applications, the **service layer** is the boundary where **business intent** is expressed and enforced. It sits between the **UI layer** (Wt widgets, events, resources, controllers) and the **data access layer** (repositories, ORM/Dbo sessions, SQL). Its purpose is not to add bureaucracy; its purpose is to keep the system **coherent**, **testable**, and **safe** as requirements grow.

15.1.1 The problem it solves

Without a service layer, business rules tend to leak into places that are not designed to hold them:

- UI callbacks and widget event handlers become large and stateful, mixing validation, authorization, DB calls, and error mapping.
- Repositories slowly become “smart” and start embedding policy decisions (which should be business logic), turning into a second business layer hidden inside persistence code.
- Cross-cutting concerns (transactions, auditing, rate limits, caching, invariants) are duplicated across endpoints, leading to inconsistent behavior.

A service layer prevents this by providing a **single, explicit home** for **use-cases** (application actions) such as:

- Register account, authenticate, change password, create order, confirm payment, cancel shipment.
- Admin workflows: approve request, lock user, recalculate invoices.
- System workflows: nightly reconciliation, import/export, scheduled cleanup.

15.1.2 What a service layer is (and is not)

It is:

- A **use-case API** for the application: methods represent business actions, not tables.
- The place to enforce **invariants** (rules that must always hold true).
- The orchestrator for **multiple repositories** and external services in one coherent transaction boundary.
- The natural integration point for cross-cutting concerns: **authZ/authN checks**, **logging**, **auditing**, **metrics**, **rate limiting**, **caching**, **idempotency**.

It is not:

- A “thin pass-through” that mirrors repository CRUD one-to-one.
- A container for UI formatting, widget code, HTML, or Wt view state.
- A dumping ground for every helper function; it should remain focused on **business actions**.

15.1.3 The key benefits in large systems

1) Clear boundaries and stable dependencies

A service layer defines strict dependency direction:

$$\text{UI (Wt)} \rightarrow \text{Services} \rightarrow \text{Repositories/DB/External}$$

This prevents the UI from depending directly on persistence details, which reduces accidental coupling and makes refactoring possible without breaking the entire codebase.

2) Business rules stay consistent

When the same rule is enforced in multiple endpoints, you get drift. The service layer centralizes rules such as:

- "A customer cannot place an order if the account is locked."
- "Discounts require eligibility and must be capped by policy."
- "A password change requires recent authentication or MFA."

Centralization ensures **uniform behavior** across all entry points (UI pages, REST resources, background jobs).

3) Transaction boundaries and orchestration

Real operations are rarely a single table update. A use-case often requires:

- multiple reads and writes,
- integrity checks,
- side effects (email, audit record, message queue),

- and controlled commit/rollback.

The service layer is the correct place to define **transaction boundaries**, ensuring atomicity where required.

4) Testability without Wt or a real database

When UI code calls repositories directly, unit tests become expensive and brittle. With services:

- the UI is tested as UI (smaller, focused tests),
- the service is tested as business logic (fast unit tests),
- and repositories are tested as persistence (integration tests).

This separation is a practical foundation for reliable CI.

5) Security and policy enforcement

Security is a business rule. Services are where you enforce:

- authorization decisions (who can do what),
- input validation rules that reflect policy (not only syntax),
- sensitive operation controls (rate limits, lockouts, step-up auth),
- audit trails for compliance.

Putting this logic in UI handlers is fragile; putting it in repositories is conceptually wrong.

15.1.4 Service layer shape in Modern C++ (pragmatic design)

Use-case oriented interfaces

Prefer interfaces that reflect intent, not CRUD. For example, a user service should not expose `insertUser()` and `updateUser()` as the primary API; instead expose actions like `registerUser()`, `authenticate()`, `changePassword()`.

```
// A lightweight "result" type: success or structured failure.
// In real code you can use std::expected (C++23) or an equivalent.
enum class ServiceError {
  InvalidInput,
  NotFound,
  Conflict,
  NotAuthorized,
  RateLimited,
  TransientFailure,
  InternalError
};

template <typename T>
struct ServiceResult {
  T value{};
  ServiceError error{ServiceError::InternalError};
  std::string message{};
  bool ok{false};

  static ServiceResult success(T v) {
    ServiceResult r;
    r.value = std::move(v);
    r.ok = true;
    return r;
  }
```

```
  static ServiceResult failure(ServiceError e, std::string m) {
    ServiceResult r;
    r.error = e;
    r.message = std::move(m);
    r.ok = false;
    return r;
  }
};
```

Keep services free of UI types

Services should avoid taking or returning `Wt::WString`, widgets, or HTTP request objects. Use plain domain types (DTOs) and keep formatting in the UI layer.

```
struct RegisterUserRequest {
  std::string email;
  std::string password;
  std::string displayName;
};

struct UserSummary {
  std::int64_t id{};
  std::string email;
  std::string displayName;
};

struct IUserService {
  virtual ~IUserService() = default;
  virtual ServiceResult<UserSummary> registerUser(const RegisterUserRequest& req) = 0;
};
```

Orchestrate repositories and enforce invariants

A service method typically:

1. validates business constraints,
2. checks policy/authorization,
3. loads required state via repositories,
4. applies changes,
5. commits transaction,
6. triggers side effects (possibly after commit),
7. maps internal failures to stable service errors.

```
// Repository interfaces are persistence-focused (tables/aggregates),
// while services are use-case focused (actions).
struct IUserRepo {
  virtual ~IUserRepo() = default;
  virtual bool emailExists(const std::string& email) = 0;
  virtual std::int64_t insertUser(std::string email, std::string hashedPw, std::string
  ↪  displayName) = 0;
};

struct IPasswordHasher {
  virtual ~IPasswordHasher() = default;
  virtual std::string hash(const std::string& pw) = 0;
};

class UserService final : public IUserService {
public:
  UserService(IUserRepo& repo, IPasswordHasher& hasher)
    : repo_(repo), hasher_(hasher) {}

  ServiceResult<UserSummary> registerUser(const RegisterUserRequest& req) override {
```

```
    if (req.email.empty() || req.password.size() < 10) {
      return ServiceResult<UserSummary>::failure(
        ServiceError::InvalidInput,
        "Email is required and password must meet the policy."
      );
    }

    // Invariant: email must be unique.
    if (repo_.emailExists(req.email)) {
      return ServiceResult<UserSummary>::failure(
        ServiceError::Conflict,
        "An account with this email already exists."
      );
    }

    const auto hashed = hasher_.hash(req.password);

    // In a real system, this insert would be done inside a transaction
    // controlled by the service (or a Unit of Work abstraction).
    const auto id = repo_.insertUser(req.email, hashed, req.displayName);

    return ServiceResult<UserSummary>::success(UserSummary{id, req.email,
    ↪  req.displayName});
  }

private:
  IUserRepo& repo_;
  IPasswordHasher& hasher_;
};
```

15.1.5 Wt-specific value: keeping widget code small and deterministic

Wt applications are event-driven: button clicks, signals, and server-side updates. If business logic is embedded inside widget callbacks, those callbacks become hard to reason about and hard to test.

A clean approach is:

- Widget collects inputs and triggers a service call.
- Service returns a stable success/failure result.
- Widget maps that result to UI messages and navigation.

```
// Pseudocode showing the boundary; the service uses std::string DTOs,
// UI remains responsible for Wt widgets and rendering.
class RegisterWidget : public Wt::WContainerWidget {
public:
  RegisterWidget(IUserService& users)
    : users_(users) {

    email_ = addNew<Wt::WLineEdit>();
    pw_    = addNew<Wt::WLineEdit>();
    pw_->setEchoMode(Wt::EchoMode::Password);

    status_ = addNew<Wt::WText>();

    auto btn = addNew<Wt::WPushButton>("Create account");
    btn->clicked().connect([this] {
      RegisterUserRequest req;
      req.email = email_->text().toUTF8();
      req.password = pw_->text().toUTF8();

      auto r = users_.registerUser(req);
```

```
      if (!r.ok) {
        status_->setText(Wt::WString::fromUTF8(r.message));
        return;
      }

      status_->setText("Account created.");
      // Navigate / update UI state here (not inside the service).
    });
  }

private:
  IUserService& users_;
  Wt::WLineEdit* email_{};
  Wt::WLineEdit* pw_{};
  Wt::WText* status_{};
};
```

15.1.6 Common anti-patterns (and why they hurt)

- **CRUD services**: exposing `create/update/delete` as your main API leads to business rules being enforced elsewhere.
- **UI-driven transactions**: opening/committing DB operations inside widget code spreads transactional behavior across the UI, causing subtle consistency bugs.
- **Repositories with policy**: repositories should not decide who is allowed to do what; they should implement persistence concerns.
- **Exceptions as API**: throwing raw DB/network exceptions across layers forces the UI to understand infrastructure failures. Services should translate failures into stable outcomes.

15.1.7 Practical checklist

- Each service method corresponds to a **use-case**, not a table.
- Services accept/return **plain DTOs**, not Wt types.
- Business invariants are enforced **once** in the service layer.
- Transaction boundaries are **owned by services** (or a Unit-of-Work controlled by them).
- Services map infrastructure issues to **stable error categories**.
- UI callbacks remain small: collect input, call service, render result.

15.1.8 Outcome

A well-designed service layer is the difference between a demo and a production system. It keeps business logic **explicit**, **auditable**, and **testable**, while allowing the UI (Wt) and persistence layer to evolve independently. When requirements expand, the service layer becomes the stable core that prevents complexity from spreading into every file of the codebase.

15.2 Isolating business rules

A production-grade application fails over time for one predictable reason: **business rules leak**. They leak into UI callbacks, SQL queries, repository methods, background jobs, and "helper" utilities until the system becomes a set of inconsistent behaviors rather than a single coherent product.

Isolating business rules means designing the code so that:

- rules are expressed **once**,

- enforced **consistently** across all entry points (Wt UI, REST endpoints, jobs),
- and tested **without** a running server or database.

In a Wt-based server-side UI, this isolation is especially important because UI code is event-driven and stateful. If business rules are implemented inside widget callbacks, they become hard to test, easy to duplicate, and easy to bypass.

15.2.1 What counts as a business rule?

A business rule is not merely "validation". It is any policy that expresses **what the system allows** and **what must always be true**.

Examples

- Eligibility policies: "Only verified users can create invoices."
- Pricing rules: "Discount cannot exceed 20% unless manager approved."
- State transitions: "Paid orders cannot be cancelled; only refunded."
- Risk control: "Lock account after N failed login attempts in 10 minutes."
- Domain invariants: "An order must have at least one line item." / "Quantity must be positive."
- Regulatory constraints: "Keep audit record for sensitive changes."

15.2.2 Why rules must be isolated

1) Consistency across multiple entry points

A single use-case can be triggered from:

- Wt pages/widgets (button click, form submit),
- an API resource,
- a scheduled job,
- an admin console.

If the rule is implemented in the UI, the job might bypass it. If it is implemented in one endpoint, another endpoint may drift. Isolation ensures **one authority**.

2) Testability and confidence

Business rules are the most valuable and fragile part of the system. They should be testable as pure logic:

- no Wt runtime required,
- no HTTP request required,
- no database required (unless a rule truly depends on persisted state).

3) Preventing "smart SQL" and policy-in-repository

SQL and repositories are excellent at persistence concerns (querying and storing data). They are poor places for policy because:

- rules become hidden in query conditions,
- logic becomes hard to reuse,
- error reporting becomes unclear,
- and the same rule gets reimplemented differently elsewhere.

15.2.3 Architecture strategy: Functional Core, Imperative Shell

A practical way to isolate business rules is to separate:

- **Functional core**: deterministic domain logic (rules, invariants, decisions).
- **Imperative shell**: I/O orchestration (DB, transactions, network, Wt UI updates).

In this model:

- The **service layer** is mostly an **imperative shell** orchestrator.
- The **domain model** (entities/value objects/policies) is the **functional core**.

15.2.4 Where to place rules: a practical taxonomy

Different rules belong in different constructs. The goal is **clarity and reuse**, not ideology.

A) Value objects for local invariants

If a rule is about the validity of a single value, express it as a **value object**. This prevents invalid states from existing in memory.

```
#include <string>
#include <string_view>
#include <cctype>

class Email {
public:
  static bool looks_valid(std::string_view s) {
    // Minimal, pragmatic check for example purposes.
    auto at = s.find('@');
    return at != std::string_view::npos && at > 0 && at + 1 < s.size();
  }
```

```
  static Email create(std::string s) {
    if (!looks_valid(s)) {
      throw std::invalid_argument("Invalid email format");
    }
    return Email(std::move(s));
  }

  const std::string& str() const noexcept { return value_; }

private:
  explicit Email(std::string v) : value_(std::move(v)) {}
  std::string value_;
};
```

Why this isolates rules Once a function accepts an `Email` instead of `std::string`, the rule cannot be forgotten or duplicated.

B) Entities/Aggregates for state invariants and transitions

If a rule depends on an entity's current state and governs transitions, it belongs in the **entity/aggregate**.

```
#include <vector>
#include <stdexcept>
#include <cstdint>

struct OrderLine {
  std::int64_t productId{};
  int quantity{};
};

enum class OrderStatus { Draft, Submitted, Paid, Cancelled, Refunded };
```

```
class Order {
public:
  void add_line(OrderLine line) {
    if (status_ != OrderStatus::Draft) {
      throw std::logic_error("Cannot modify order after submission");
    }
    if (line.quantity <= 0) {
      throw std::invalid_argument("Quantity must be positive");
    }
    lines_.push_back(line);
  }

  void submit() {
    if (status_ != OrderStatus::Draft) {
      throw std::logic_error("Only draft orders can be submitted");
    }
    if (lines_.empty()) {
      throw std::logic_error("Order must contain at least one line");
    }
    status_ = OrderStatus::Submitted;
  }

  void mark_paid() {
    if (status_ != OrderStatus::Submitted) {
      throw std::logic_error("Only submitted orders can be paid");
    }
    status_ = OrderStatus::Paid;
  }

  void cancel() {
    if (status_ == OrderStatus::Paid) {
      throw std::logic_error("Paid orders cannot be cancelled; refund instead");
```

```
    }
    if (status_ == OrderStatus::Cancelled) {
      return;
    }
    status_ = OrderStatus::Cancelled;
  }

  OrderStatus status() const noexcept { return status_; }

private:
  OrderStatus status_{OrderStatus::Draft};
  std::vector<OrderLine> lines_{};
};
```

Why this isolates rules No UI endpoint can accidentally "forget" a transition rule because the state changes are guarded by the domain object itself.

C) Policy/Specification objects for reusable decisions

When a rule is a reusable decision (eligibility, limits, compliance), implement it as a **policy** (or a **specification**) that can be composed and tested independently.

```
#include <chrono>
#include <cstdint>

struct UserAccount {
  bool verified{false};
  bool locked{false};
  int failedLoginsInWindow{0};
};

class LoginPolicy {
public:
```

```
  explicit LoginPolicy(int maxFails) : maxFails_(maxFails) {}

  bool may_attempt_login(const UserAccount& u) const noexcept {
    if (u.locked) return false;
    return u.failedLoginsInWindow < maxFails_;
  }

private:
  int maxFails_{5};
};
```

Why this isolates rules The service layer can reuse the same policy in UI login, API login, and background security enforcement.

D) Domain services for rules that span aggregates

If a rule requires coordination across multiple domain objects (and cannot naturally live on a single entity), use a **domain service**. It should remain pure logic as much as possible.

```
#include <cstdint>

struct Money {
  std::int64_t cents{};
};

class DiscountPolicy {
public:
  // Example: cap discount to 20% unless manager override
  Money apply_discount(Money subtotal, int percent, bool managerApproved) const {
    if (percent < 0) percent = 0;
    if (!managerApproved && percent > 20) percent = 20;

    // cents math to avoid floating point issues
```

```
    const auto discount = (subtotal.cents * percent) / 100;
    return Money{subtotal.cents - discount};
  }
};
```

15.2.5 Service layer role: orchestration, not rule storage

The service layer should **call** rules (domain objects/policies), not **contain** them. Its responsibilities:

- gather state (load aggregates),
- enforce authorization at the use-case boundary,
- start/commit/rollback transactions,
- persist changes,
- publish side effects (audit, email, messaging),
- map errors into stable outcomes.

Use-case result types: stable and UI-friendly

Avoid leaking infrastructure exceptions as part of the public use-case API. Return stable outcomes that the UI can map to messages.

```
#include <string>
#include <utility>

enum class UseCaseError {
  InvalidInput,
  NotAuthorized,
  Conflict,
```

```
  NotFound,
  RuleViolation,
  TransientFailure,
  InternalError
};

template <class T>
struct UseCaseResult {
  bool ok{false};
  T value{};
  UseCaseError error{UseCaseError::InternalError};
  std::string message{};

  static UseCaseResult success(T v) {
    UseCaseResult r; r.ok = true; r.value = std::move(v); return r;
  }
  static UseCaseResult fail(UseCaseError e, std::string m) {
    UseCaseResult r; r.ok = false; r.error = e; r.message = std::move(m); return r;
  }
};
```

Service orchestration example: isolate rules, centralize use-case

```
#include <stdexcept>
#include <cstdint>
#include <string>

// Abstractions: persistence and external services (ports)
struct IOrderRepo {
  virtual ~IOrderRepo() = default;
  virtual Order load(std::int64_t orderId) = 0;
  virtual void save(const Order& o) = 0;
};
```

```
struct IAuthorization {
  virtual ~IAuthorization() = default;
  virtual bool can_submit_order(std::int64_t userId, std::int64_t orderId) = 0;
};

// DTO returned to UI (keep it small and stable)
struct OrderSummary {
  std::int64_t orderId{};
  OrderStatus status{};
};

class OrderService {
public:
  OrderService(IOrderRepo& repo, IAuthorization& auth)
    : repo_(repo), auth_(auth) {}

  UseCaseResult<OrderSummary> submit_order(std::int64_t userId, std::int64_t orderId) {
    if (!auth_.can_submit_order(userId, orderId)) {
      return UseCaseResult<OrderSummary>::fail(
        UseCaseError::NotAuthorized, "Not allowed to submit this order."
      );
    }

    try {
      Order o = repo_.load(orderId);

      // Business rules are inside the aggregate:
      // - must be Draft
      // - must have at least one line
      o.submit();

      repo_.save(o);
```

```
      return UseCaseResult<OrderSummary>::success(OrderSummary{orderId, o.status()});
    } catch (const std::invalid_argument& e) {
      return UseCaseResult<OrderSummary>::fail(UseCaseError::InvalidInput, e.what());
    } catch (const std::logic_error& e) {
      return UseCaseResult<OrderSummary>::fail(UseCaseError::RuleViolation, e.what());
    } catch (const std::exception&) {
      // In real code, log details internally; keep API stable.
      return UseCaseResult<OrderSummary>::fail(
        UseCaseError::InternalError, "Operation failed."
      );
    }
  }

private:
  IOrderRepo& repo_;
  IAuthorization& auth_;
};
```

What this achieves

- The rule "order must contain at least one line" lives in `Order::submit()`.
- Any entry point calling `submit_order()` gets identical behavior.
- The UI never needs to know how the rule is implemented; it only reacts to `UseCaseResult`.

15.2.6 Preventing rule duplication in the UI (Wt)

Wt widgets should be a thin adapter: collect user input, call the service, render the result. **Do not replicate rules** in UI (except for UX-only hints such as immediate field feedback).

```
#include <Wt/WText.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WContainerWidget.h>

class SubmitOrderWidget : public Wt::WContainerWidget {
public:
  SubmitOrderWidget(OrderService& svc, std::int64_t userId)
    : svc_(svc), userId_(userId) {

    orderId_ = addNew<Wt::WLineEdit>();
    status_  = addNew<Wt::WText>();

    auto btn = addNew<Wt::WPushButton>("Submit");
    btn->clicked().connect([this] {
      const auto idStr = orderId_->text().toUTF8();
      const auto orderId = static_cast<std::int64_t>(std::stoll(idStr));

      auto r = svc_.submit_order(userId_, orderId);
      if (!r.ok) {
        status_->setText(Wt::WString::fromUTF8(r.message));
        return;
      }
      status_->setText("Submitted.");
    });
  }

private:
  OrderService& svc_;
  std::int64_t userId_{};
  Wt::WLineEdit* orderId_{};
  Wt::WText* status_{};
};
```

15.2.7 Techniques that strengthen rule isolation

1) “Invalid states are unrepresentable” (strong types)

Prefer domain types over primitive obsession. Use:

- value objects (Email, Money, Quantity),
- scoped enums for state,
- dedicated IDs (UserId, OrderId) to prevent mix-ups.

```
#include <cstdint>

struct UserId { std::int64_t value{}; };
struct OrderId { std::int64_t value{}; };

inline bool operator==(UserId a, UserId b) { return a.value == b.value; }
inline bool operator==(OrderId a, OrderId b) { return a.value == b.value; }
```

2) Keep I/O out of rule code

Business rules should not perform:

- SQL queries,
- HTTP calls,
- filesystem access,
- widget operations.

Instead, gather required data in the service layer, then feed it into the domain/policy logic.

3) Centralize rule evaluation for cross-cutting policies

Rules like rate limits, lockouts, and compliance checks should be enforced at the use-case boundary so they cannot be bypassed.

4) Stable error vocabulary

Define a small set of error categories (InvalidInput, NotAuthorized, RuleViolation, Conflict, NotFound, TransientFailure, InternalError). Map internal exceptions and DB errors into this vocabulary. This prevents UI code from coupling to infrastructure detail.

5) Design tests around the rule core

Your highest-value tests should target:

- aggregates and their transitions,
- policies/specifications,
- domain services.

Service tests then confirm orchestration (transaction boundaries, repository calls, correct mapping).

15.2.8 Common anti-patterns (and how to avoid them)

- **Rule in SQL**: `WHERE status != 'PAID'` appears in multiple places; eventually one query forgets it.
- **Rule in UI**: Wt callback checks something; background job bypasses it.
- **Rule in repository**: repository method silently enforces policy; service layer becomes unaware and inconsistent.

- **Duplicated validation**: UI enforces one set, service enforces another; users hit contradictions.

15.2.9 Checklist for a clean isolation

- Use-case entry points call **services**.
- Services load state and call **domain rules** (entities/policies).
- Entities/policies enforce invariants and transitions **in memory**.
- Repositories persist state; they do not embed policy.
- UI renders results; it does not own business decisions.
- Rules are unit-tested without Wt and without the database.

15.2.10 Outcome

When business rules are isolated, you gain more than "clean code". You gain:

- consistent behavior across the entire system,
- maintainable growth as requirements evolve,
- safer refactoring,
- and test suites that protect the product's real value: its policies and decisions.

15.3 Improving testability

Testability is not a "testing topic"; it is an **architecture property**. In production-grade server-side systems, the main reason test suites become slow, flaky, or incomplete is that the codebase is shaped so that meaningful behavior cannot be exercised without:

- a running server,
- a live database,
- a configured Wt session,
- and real external dependencies.

A service layer improves testability by making **business behavior callable as plain C++** with controlled dependencies. The service API becomes the stable unit under test, while UI, DB, and integrations become adapters that can be mocked, faked, or tested separately.

15.3.1 What we mean by "testability" in a service-layer design

A testable service layer provides:

- **Determinism**: the same input state produces the same output result.
- **Isolation**: business rules can be tested without I/O and without Wt widgets.
- **Observability**: outcomes are returned as explicit results or domain state, not hidden in logs.
- **Speed**: unit tests complete quickly and can run frequently.
- **Composability**: tests can cover use-cases end-to-end at the service boundary without expensive integration setup.

15.3.2 How the service layer improves testability

1) Separating imperative I/O from rule evaluation

Service methods orchestrate I/O, but **rules belong in the domain core** (entities, value objects, policy classes).

- The service layer pulls required data using repositories.
- It calls pure rule logic in memory.
- It persists changes and triggers side effects.

This structure lets you unit-test the rule logic independently (fast) and service orchestration with fakes/mocks (still fast), while reserving DB tests for a smaller integration suite.

2) Explicit dependency injection (DI) via constructor parameters

Testable code avoids hidden global dependencies. Services should accept:

- repositories via interfaces,
- clocks/time providers,
- random generators,
- password hashers/crypto,
- message senders,
- transaction/unit-of-work abstractions.

Constructor injection allows tests to substitute deterministic doubles.

3) Stable outcomes instead of infrastructure exceptions

UI and tests should not depend on DB-specific or network-specific exception types. The service layer should map failures into a stable error vocabulary.

```
#include <string>
#include <utility>
```

```
enum class UseCaseError {
  InvalidInput,
  NotAuthorized,
  NotFound,
  Conflict,
  RuleViolation,
  TransientFailure,
  InternalError
};

template <class T>
struct Result {
  bool ok{false};
  T value{};
  UseCaseError error{UseCaseError::InternalError};
  std::string message{};

  static Result success(T v) {
    Result r; r.ok = true; r.value = std::move(v); return r;
  }
  static Result fail(UseCaseError e, std::string m) {
    Result r; r.ok = false; r.error = e; r.message = std::move(m); return r;
  }
};
```

Tests can assert:

- success value,
- failure category,
- and message (if you choose to keep it stable).

15.3.3 Designing services for testability: concrete patterns

Pattern A: Port & Adapter interfaces (repositories and side effects)

Define minimal interfaces for anything that talks to the outside world. This creates "ports" the service uses.

```
#include <cstdint>
#include <optional>
#include <string>

struct User {
  std::int64_t id{};
  std::string email;
  bool locked{false};
  int failedLoginsInWindow{0};
};

struct IUserRepo {
  virtual ~IUserRepo() = default;
  virtual std::optional<User> find_by_email(const std::string& email) = 0;
  virtual void update(const User& u) = 0;
};

struct IAuditLog {
  virtual ~IAuditLog() = default;
  virtual void record(std::int64_t userId, std::string_view action) = 0;
};
```

In tests, you can provide:

- an in-memory `FakeUserRepo`,
- a `NullAuditLog` or `SpyAuditLog`.

Pattern B: Time control with a Clock abstraction

Time-based logic is a classic source of flaky tests. Use an injected clock.

```
#include <chrono>

struct IClock {
  virtual ~IClock() = default;
  virtual std::chrono::system_clock::time_point now() const = 0;
};

class SystemClock final : public IClock {
public:
  std::chrono::system_clock::time_point now() const override {
    return std::chrono::system_clock::now();
  }
};

class FakeClock final : public IClock {
public:
  explicit FakeClock(std::chrono::system_clock::time_point t) : t_(t) {}
  std::chrono::system_clock::time_point now() const override { return t_; }
  void advance(std::chrono::seconds s) { t_ += s; }
private:
  std::chrono::system_clock::time_point t_;
};
```

Now lockout windows, token expirations, and rate limits become deterministic in tests.

Pattern C: Transaction boundaries that are test-friendly

In production, a service usually wraps a use-case in a transaction. In tests, you often want the same interface without a real DB.
A simple approach is a **Unit of Work** port with a test fake.

```
#include <functional>

struct IUnitOfWork {
  virtual ~IUnitOfWork() = default;
  virtual void run_in_transaction(const std::function<void()>& fn) = 0;
};

class NoopUnitOfWork final : public IUnitOfWork {
public:
  void run_in_transaction(const std::function<void()>& fn) override {
    fn(); // For unit tests: no real transaction, but same control flow.
  }
};
```

This keeps service code consistent between production and test environments.

15.3.4 Example: a testable service use-case

Service implementation

This example demonstrates:

- rule checks in memory,
- repository calls abstracted,
- audit side effect abstracted,
- deterministic lockout logic.

```
#include <string>
#include <optional>

class AuthService {
```

```
public:
  AuthService(IUserRepo& users, IAuditLog& audit, IClock& clock, IUnitOfWork& uow)
    : users_(users), audit_(audit), clock_(clock), uow_(uow) {}

  Result<User> attempt_login(const std::string& email, const std::string& password) {
    if (email.empty() || password.empty()) {
      return Result<User>::fail(UseCaseError::InvalidInput,
      ↪  "Email and password are required.");
    }

    auto opt = users_.find_by_email(email);
    if (!opt) {
      // Avoid information leakage; return stable failure.
      return Result<User>::fail(UseCaseError::NotFound, "Invalid credentials.");
    }

    User u = *opt;

    if (u.locked) {
      return Result<User>::fail(UseCaseError::NotAuthorized, "Account is locked.");
    }

    // For example purposes only: password verification omitted.
    const bool ok = (password == "demo"); // Replace with injected hasher/verifier.

    uow_.run_in_transaction([&] {
      if (!ok) {
        u.failedLoginsInWindow += 1;
        if (u.failedLoginsInWindow >= 5) {
          u.locked = true;
          audit_.record(u.id, "account_locked");
        }
        users_.update(u);
```

```
      } else {
        // On successful login you might reset counters, update last-login time, etc.
        u.failedLoginsInWindow = 0;
        users_.update(u);
        audit_.record(u.id, "login_success");
      }
    });

    if (!ok) {
      return Result<User>::fail(UseCaseError::NotAuthorized, "Invalid credentials.");
    }

    return Result<User>::success(u);
  }

private:
  IUserRepo& users_;
  IAuditLog& audit_;
  IClock& clock_;
  IUnitOfWork& uow_;
};
```

Test doubles: Fake repository and audit spy

```
#include <vector>
#include <optional>
#include <string_view>

class FakeUserRepo final : public IUserRepo {
public:
  std::optional<User> find_by_email(const std::string& email) override {
    for (const auto& u : users_) {
      if (u.email == email) return u;
```

```
    }
    return std::nullopt;
  }

  void update(const User& u) override {
    for (auto& x : users_) {
      if (x.id == u.id) { x = u; return; }
    }
    users_.push_back(u);
  }

  std::vector<User> users_{};
};

class SpyAuditLog final : public IAuditLog {
public:
  void record(std::int64_t userId, std::string_view action) override {
    events.emplace_back(userId, std::string(action));
  }
  std::vector<std::pair<std::int64_t, std::string>> events{};
};
```

Unit test example (framework-agnostic style)

The goal is to show assertions clearly without binding your book to a single test framework.

```
#include <cassert>
#include <chrono>

static void test_lock_after_five_failures() {
  FakeUserRepo repo;
  repo.users_.push_back(User{1, "a@b.com", false, 4});

  SpyAuditLog audit;
```

```
  auto t0 = std::chrono::system_clock::time_point{};
  FakeClock clock(t0);
  NoopUnitOfWork uow;

  AuthService svc(repo, audit, clock, uow);

  auto r = svc.attempt_login("a@b.com", "wrong");
  assert(!r.ok);
  assert(r.error == UseCaseError::NotAuthorized);

  auto u = repo.find_by_email("a@b.com");
  assert(u.has_value());
  assert(u->locked == true);
  assert(!audit.events.empty());
  assert(audit.events.back().second == "account_locked");
}
```

What this demonstrates

- The service can be tested without Wt, without HTTP, and without a database.
- The test is deterministic and fast.
- Side effects are observable through the spy.
- The use-case behavior is asserted at the service boundary.

15.3.5 Testing strategy: layering your tests correctly

A production-grade project typically uses three complementary test levels.

1) Pure domain tests (fastest)

Test entities, value objects, and policies in isolation:

- invariants,
- state transitions,
- calculations,
- eligibility decisions.

These tests should run in milliseconds and cover the majority of business logic.

2) Service tests with fakes/mocks (fast)

Test orchestration:

- correct repository calls,
- correct transaction usage,
- correct mapping of errors,
- correct side-effect triggering.

3) Integration tests (slower, smaller set)

Use a real DB and real wiring to validate:

- SQL schema constraints match domain expectations,
- repository implementations,
- transaction isolation behavior,
- migrations and real queries.

Keep these fewer and focused; they are essential but expensive.

15.3.6 Wt-specific guidance for testability

Wt is excellent for server-side UI, but UI tests can be heavy if you try to test everything through widgets. A practical approach:

- Put business logic behind services and test services.
- Keep widget callbacks thin:
  - read fields,
  - call service,
  - render result,
  - update navigation/state.
- Do not embed DB calls or complex rule logic inside widget event handlers.

15.3.7 Common pitfalls that destroy testability

- **Hidden singletons/globals**: cannot be controlled or substituted in tests.
- **Time calls scattered everywhere**: `now()` in many places causes non-determinism.
- **Exceptions as control flow**: tests become dependent on infrastructure exception types.
- **Fat UI callbacks**: forcing tests to boot Wt sessions to validate business behavior.
- **Mixing formatting with rules**: services returning UI-ready strings rather than stable results.

15.3.8 Practical checklist

- Services depend on interfaces (repos, audit, mail, clock, unit-of-work).
- Business rules live in domain types (entities/value objects/policies), not in UI or SQL.
- Services return stable result types with clear error categories.
- Time is injected and controllable.
- Side effects are abstracted and observable (spy/fake).
- Integration tests validate infrastructure; unit tests validate behavior.

15.3.9 Outcome

When the service layer is designed for testability, you gain the ability to evolve business logic rapidly without fear:

- rule changes become local and verifiable,
- regressions are caught early,
- Wt UI remains clean and maintainable,
- and the database becomes an implementation detail rather than a testing dependency.

Controllers

16.1 The role of controllers in Wt

In Modern C++ web applications built with the Wt toolkit, **controllers** play a distinct role as part of the Business Logic Layer. While Wt combines UI and HTTP handling into widgets and resources, adopting a controller abstraction introduces clarity, separation of concerns, and testability for request handling, routing, and use-case orchestration.
This section explains what controllers are, why they matter in Wt applications, how they interact with the service layer and views, and how to design them for production-grade systems.

16.1.1 What is a controller?

A **controller** is an abstraction that:

- receives an HTTP request or route invocation,
- extracts input parameters,
- performs authorization and validation,
- calls appropriate services (business use-cases),
- handles errors and maps them to HTTP responses,

- and chooses the output representation (JSON, HTML, redirect).

Controllers are not business logic holders; they are **orchestrators at the edge of the request-response cycle**.

16.1.2 Why controllers matter in Wt

Separation of concerns

Wt widgets and resources excel at building interactive UI and managing DOM-like behavior. However, mixing request parsing, business use-case invocation, and response mapping into widgets leads to:

- difficult-to-test event handlers,
- duplicated parsing and error handling,
- scattered authorization checks,
- and brittle endpoint behavior.

Controllers encapsulate:

- routing and request binding,
- input coercion and validation,
- service invocation,
- response mapping,
- and HTTP status control.

This makes widgets simpler and controllers the canonical place for endpoint logic.

Consistent error and response handling

Controllers centralize:

- mapping service errors into HTTP status codes,
- uniform JSON or HTML error structures,
- content negotiation,
- and success response formatting.

This consistency is essential for maintainable REST APIs and predictable web interactions.

Testability

By moving HTTP-specific handling into controllers with service dependencies injected, you can:

- test routing logic without booting the full Wt server stack,
- mock services and verify controller behavior in isolation,
- assert status codes and response bodies.

Widgets then focus on presentation, not business decisions.

16.1.3 Controllers vs. Wt resources and widgets

- **Wt Widgets**: server-side UI components with event-driven behavior for interactive pages.
- **Wt Resource**: a class handling HTTP requests directly (often for REST endpoints).

- **Controller**: a higher-level orchestrator that may use Wt resources or routing, mapping requests to services and responses.

In many designs:

- a `Controller` handles routes and delegates to services,
- Wt supports routing via `Wt::WApplication::addResource` or `Wt::WEnvironment`,
- controllers avoid UI state and focus on request/response logic.

16.1.4 Design principles for Wt controllers

1) Controllers depend on services, not persistence

Controllers should:

- accept service interfaces via constructor injection,
- call service use-cases,
- handle use-case results,
- and map them to HTTP outcomes.

They should not call repositories or database code directly.

2) Controllers separate HTTP concerns from domain logic

Controllers should extract and validate:

- path parameters,
- query parameters,

- headers,
- and request bodies (JSON, form data).

After validation, a controller invokes services, not business rules directly.

3) Use result types for predictable mapping

Controllers should interpret stable result types from services (e.g., success/failure with error enums) and map them to appropriate HTTP codes.

16.1.5 Example: a REST controller for user registration

DTOs and service interface

```
struct RegisterUserRequest {
  std::string email;
  std::string password;
  std::string displayName;
};

struct UserSummary {
  std::int64_t id;
  std::string email;
  std::string displayName;
};

struct IUserService {
  virtual ~IUserService() = default;
  virtual Result<UserSummary> registerUser(const RegisterUserRequest&) = 0;
};
```

JSON parsing utility

A minimal helper to parse JSON bodies:

```
#include <Wt/Http/Request.h>
#include <Wt/Json/Parser.h>
#include <Wt/Json/Object.h>

std::optional<RegisterUserRequest> parse_register_json(const Wt::Http::Request& req) {
  if (req.contentType() != "application/json") return std::nullopt;
  const auto body = req.body();
  Wt::Json::Parser parser;
  auto json = parser.parse(body);
  if (!json || !json->isObject()) return std::nullopt;
  auto obj = json->toObject();
  RegisterUserRequest r;
  r.email = obj.get("email").orIfNull("").toString();
  r.password = obj.get("password").orIfNull("").toString();
  r.displayName = obj.get("displayName").orIfNull("").toString();
  return r;
}
```

Controller implementation

```
#include <Wt/Http/Response.h>
#include <Wt/Http/Request.h>
#include <string_view>

class UserController {
public:
  UserController(IUserService& svc)
    : users_(svc) {}

  void handle_register(const Wt::Http::Request& req, Wt::Http::Response& resp) {
```

```
    auto maybeReq = parse_register_json(req);
    if (!maybeReq) {
      resp.setStatus(400);
      resp.out() << R"({"error":"Invalid JSON"})";
      return;
    }

    auto result = users_.registerUser(*maybeReq);
    if (!result.ok) {
      switch (result.error) {
        case UseCaseError::InvalidInput:
          resp.setStatus(422);
          break;
        case UseCaseError::Conflict:
          resp.setStatus(409);
          break;
        default:
          resp.setStatus(500);
          break;
      }
      resp.out() << R"({"error":")" << result.message << R"("})";
      return;
    }

    resp.setStatus(201);
    resp.out() << R"({"id":)" << result.value.id
               << R"(,"email":")" << result.value.email
               << R"(","displayName":")" << result.value.displayName << "\"}";
  }

private:
  IUserService& users_;
};
```

Registering the controller with Wt

```
#include <Wt/WServer.h>
#include <Wt/Http/Response.h>
#include <memory>

int main(int argc, char** argv) {
  Wt::WServer server{"MyApp", argc, argv};

  // Construct services and controllers
  auto userService = std::make_shared<MyUserService>();
  auto userCtrl    = std::make_shared<UserController>(*userService);

  // Bind REST endpoint
  server.addResource(
    std::make_shared<Wt::WResource>(
      [userCtrl](const Wt::Http::Request& req, Wt::Http::Response& resp) {
        userCtrl->handle_register(req, resp);
      }
    ),
    "/api/users"
  );

  server.run();
}
```

16.1.6 Routing and controllers

Wt supports flexible routing. Controllers should be agnostic to specific routing mechanisms. Use server-side routing definitions to map URIs to controller methods.

Example: parameterized route

```
server.addResource(
  std::make_shared<Wt::WResource>(
    [userCtrl](const Wt::Http::Request& req, Wt::Http::Response& resp) {
      const auto idStr = req.pathInfo().substr(1);
      const auto userId = std::stoll(idStr);
      userCtrl->handle_get(userId, req, resp);
    }
  ),
  "/api/users/"
);
```

Controllers should handle invalid parameter values gracefully and return appropriate HTTP statuses.

16.1.7 Controller responsibilities in detail

Input extraction and validation Controllers parse:

- path segments,
- query parameters,
- headers,
- and body formats (JSON, form-encoded).

Controllers should validate that inputs meet syntactic requirements before forwarding to services.

Authorization and authentication Controllers often perform:

- authentication checks (sessions, tokens, cookies),

- authorization decisions (guards),
- and reject unauthorized requests early.

Authorization decisions should be delegated to policies or services; controllers only enforce them at the HTTP boundary.

Error handling and response mapping Controllers inspect the outcome of services and map:

- domain or use-case errors to proper HTTP statuses,
- success values to structured representations,
- and unexpected errors to generic server error responses.

Content negotiation Controllers may inspect `Accept` headers to decide:

- JSON responses,
- HTML views,
- or other formats (XML, CSV).

This centralizes output choice instead of scattering it through widgets.

16.1.8 Testing controllers

Because controllers encapsulate HTTP logic, they can be tested by:

- constructing fake request/response objects,
- injecting mocked service interfaces,

- asserting status codes and bodies,
- without running the full Wt server.

```
#include <cassert>

void test_register_bad_json() {
  FakeRequest req{"POST", "/api/users", "text/plain", ""};
  FakeResponse resp;
  UserController ctrl(fakeUserService);

  ctrl.handle_register(req, resp);

  assert(resp.status() == 400);
  assert(resp.body() == R"({"error":"Invalid JSON"})");
}
```

16.1.9 Best practices for controllers in Wt applications

- Keep controllers thin: delegate business decisions to services.
- Validate inputs early and reject invalid requests promptly.
- Map service results to HTTP statuses consistently.
- Avoid embedding HTML/UI logic in controllers; send data or choose views.
- Use dependency injection for services to enable testing.
- Log pertinent events at the controller boundary (requests, errors, critical decisions).

16.1.10 Outcome

Well-designed controllers in Wt applications:

- bridge HTTP and business logic cleanly,
- enhance maintainability and clarity,
- centralize input/output concerns,
- and enable high test coverage without requiring a full server stack.

Controllers become the backbone of request handling, allowing the rest of the systemservices, domain logic, viewsto remain focused, decoupled, and robust.

16.2 Coordinating between UI and services

In a production-grade Wt application, the controller layer is the **coordination boundary** between:

- the **UI layer** (Wt widgets, forms, signals/slots, navigation),
- and the **service layer** (use-cases, business rules, transactions, policies).

This coordination is not an implementation detail. It is the main mechanism by which you keep your UI responsive, your business logic consistent, and your system testable. The controller is the place where you connect **user intent** to **business intent** without allowing either side to contaminate the other.

16.2.1 The core objective

The objective of coordination is simple:

> **UI gathers user input and displays results. Services enforce rules and perform business actions. Controllers translate between the two.**

This translation includes:

- mapping UI inputs into service DTOs,
- invoking use-cases,
- mapping service results into UI-friendly view models,
- deciding UI navigation and application flow,
- applying cross-cutting concerns at the boundary (auth, logging, rate limits),
- and ensuring that UI callbacks remain small and deterministic.

16.2.2 Why this separation matters in Wt

Wt is server-side and event-driven:

- Widgets emit signals on user actions.
- Code executes on the server.
- UI updates are synchronized back to the browser (AJAX/WebSocket-style updates).

Without a controller boundary, it is common to embed:

- validation policy,
- authorization checks,
- transaction control,
- repository calls,
- and error mapping

directly into widget callbacks. That makes the UI layer:

- hard to test (requires Wt runtime),
- error-prone (logic duplicated across widgets/pages),
- and inconsistent (some flows bypass checks).

The controller prevents this by acting as the **single orchestration point** per interaction flow.

16.2.3 Controller responsibilities when coordinating UI and services

A controller typically owns the coordination steps below.

1) Input acquisition and normalization

Controllers extract values from the UI:

- text fields, combos, checkboxes,
- selected IDs,
- uploaded files,
- navigation state and route parameters.

Then they normalize inputs before calling services:

- trimming whitespace,
- converting types safely,
- converting Wt strings to UTF-8 for DTOs,
- performing syntactic validation (not business policy).

2) Building service requests (DTO mapping)

Controllers build DTOs that represent business intent. These DTOs should not contain Wt types or widget pointers.

3) Invoking use-cases

Controllers call service methods and treat the service API as the authoritative source of business outcomes.

4) Mapping outcomes to UI

Controllers map service outcomes to UI states:

- success: show confirmation, refresh view, redirect, update model.
- failure: show field errors, display global message, stay on page.
- transient failure: show retry guidance, keep UI consistent.

5) Flow control and navigation

Controllers decide:

- which view to show next,
- which part of UI updates dynamically,
- whether a redirect or internal path change is needed.

6) Cross-cutting enforcement at the boundary

Controllers are a practical place for:

- authentication/authorization gating at entry points,

- rate-limit guards for UI actions,
- consistent logging of important interactions,
- structured error translation for user-visible messages.

16.2.4 A robust pattern: View → Controller → Service

The pattern is:

1. A Wt widget collects input and triggers an event.
2. The widget calls a controller method with raw inputs.
3. The controller builds a DTO and calls the service.
4. The service executes rules, transactions, persistence, and returns a stable result.
5. The controller translates the result into UI updates.

16.2.5 Stable service result types enable clean coordination

The cleanest UI/service coordination happens when services return stable result types instead of throwing infrastructure exceptions to the UI.

```
#include <string>
#include <utility>

enum class UseCaseError {
  InvalidInput,     // syntactic or policy validation failure
  NotAuthorized,    // user cannot perform action
  NotFound,         // referenced entity missing
  Conflict,         // optimistic concurrency / uniqueness / state conflict
  RuleViolation,    // domain invariant violated
```

```
  TransientFailure, // retryable: network/DB/timeouts
  InternalError     // unexpected failure
};

template <class T>
struct Result {
  bool ok{false};
  T value{};
  UseCaseError error{UseCaseError::InternalError};
  std::string message{};

  static Result success(T v) {
    Result r; r.ok = true; r.value = std::move(v); return r;
  }
  static Result fail(UseCaseError e, std::string m) {
    Result r; r.ok = false; r.error = e; r.message = std::move(m); return r;
  }
};
```

With such results:

- UI and controller code become simple and predictable.
- Error mapping is consistent across the application.

16.2.6 Example: Registration flow coordinated by a controller

Service contracts

```
#include <cstdint>
#include <string>

struct RegisterUserRequest {
  std::string email;
```

```
  std::string password;
  std::string displayName;
};

struct UserSummary {
  std::int64_t id{};
  std::string email;
  std::string displayName;
};

struct IUserService {
  virtual ~IUserService() = default;
  virtual Result<UserSummary> register_user(const RegisterUserRequest& req) = 0;
};
```

Controller: translating UI input into a use-case

```
#include <Wt/WString.h>
#include <string>

struct RegisterUiFields {
  Wt::WString email;
  Wt::WString password;
  Wt::WString displayName;
};

class AccountController {
public:
  explicit AccountController(IUserService& users)
    : users_(users) {}

  Result<UserSummary> register_from_ui(const RegisterUiFields& ui) {
    RegisterUserRequest req;
```

```
    req.email = ui.email.toUTF8();
    req.password = ui.password.toUTF8();
    req.displayName = ui.displayName.toUTF8();

    // UI-side normalization: trim only (no business policy here).
    trim_in_place(req.email);
    trim_in_place(req.displayName);

    // Delegate the real business decision to the service.
    return users_.register_user(req);
  }

private:
  static void trim_in_place(std::string& s) {
    auto is_space = [](unsigned char c){ return c == ' ' || c == '\t' || c == '\n' || c ==
    ↪  '\r'; };
    while (!s.empty() && is_space(static_cast<unsigned char>(s.front())))
    ↪  s.erase(s.begin());
    while (!s.empty() && is_space(static_cast<unsigned char>(s.back())))  s.pop_back();
  }

  IUserService& users_;
};
```

Widget: UI concerns only

```
#include <Wt/WContainerWidget.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class RegisterPage : public Wt::WContainerWidget {
public:
```

```
  RegisterPage(AccountController& ctrl)
    : ctrl_(ctrl) {

    email_ = addNew<Wt::WLineEdit>();
    pass_  = addNew<Wt::WLineEdit>();
    name_  = addNew<Wt::WLineEdit>();
    pass_->setEchoMode(Wt::EchoMode::Password);

    status_ = addNew<Wt::WText>();

    auto btn = addNew<Wt::WPushButton>("Create account");
    btn->clicked().connect([this] {
      RegisterUiFields ui;
      ui.email = email_->text();
      ui.password = pass_->text();
      ui.displayName = name_->text();

      auto r = ctrl_.register_from_ui(ui);

      if (!r.ok) {
        // UI decides presentation; controller/service decide semantics.
        status_->setText(Wt::WString::fromUTF8(r.message));
        return;
      }

      status_->setText("Account created.");
      // Navigation decisions can be centralized in controller if preferred.
      // For example: go to login page, or redirect to profile.
    });
  }

private:
  AccountController& ctrl_;
```

```
  Wt::WLineEdit* email_{};
  Wt::WLineEdit* pass_{};
  Wt::WLineEdit* name_{};
  Wt::WText* status_{};
};
```

What this design guarantees

- The widget contains no DB access, no business rule logic.
- The controller performs safe input mapping and orchestration.
- The service enforces rules and returns stable outcomes.
- The UI simply renders a result.

16.2.7 Handling dynamic UI updates without leaking business logic

Wt encourages partial updates. The controller pattern keeps those updates clean:

- UI triggers an action (e.g., “Apply coupon”).
- Controller calls a service that validates and computes a new price.
- Controller returns a view model.
- UI updates only the affected widgets (price label, warning message).

```
#include <cstdint>
#include <string>

struct ApplyCouponRequest {
  std::int64_t cartId{};
  std::string code;
```

```
};

struct CartTotalsView {
  std::int64_t subtotalCents{};
  std::int64_t discountCents{};
  std::int64_t totalCents{};
};

struct ICartService {
  virtual ~ICartService() = default;
  virtual Result<CartTotalsView> apply_coupon(const ApplyCouponRequest&) = 0;
};

class CartController {
public:
  explicit CartController(ICartService& svc) : svc_(svc) {}

  Result<CartTotalsView> apply_coupon_from_ui(std::int64_t cartId, const Wt::WString& code)
  ↪  {
    ApplyCouponRequest req;
    req.cartId = cartId;
    req.code = code.toUTF8();
    return svc_.apply_coupon(req);
  }

private:
  ICartService& svc_;
};
```

16.2.8 Avoiding common coordination mistakes

Mistake 1: UI calls repositories directly

This causes:

- duplicated transactions,
- inconsistent policy checks,
- and untestable UI flows.

Mistake 2: Controllers contain business rules

Controllers should not decide:

- whether a discount is allowed,
- whether a user is eligible,
- whether an order can be cancelled.

Controllers may validate syntax and required fields, but business eligibility belongs to services/domain.

Mistake 3: Services return UI-oriented strings

Services should return stable data and error categories, not HTML or UI-specific messages. If you want user-facing messages, keep them minimal and stable or translate them in controllers.

Mistake 4: Throwing raw exceptions across layers

Raw DB/network exceptions leaking into UI cause:

- unpredictable UI behavior,

- framework coupling,
- and fragile error handling.

Instead, map exceptions into stable service results.

16.2.9 A practical checklist for controller coordination

- Widget callbacks do: collect inputs → call controller → render result.
- Controller does: normalize inputs → build DTO → call service → map to view model.
- Service does: enforce rules → transactions → persistence → side effects.
- No Wt widget types are allowed inside services.
- No repository/SQL access is allowed inside UI widgets.
- Service results have stable error categories that controllers map consistently.

16.2.10 Outcome

When controllers coordinate properly between UI and services, the architecture gains:

- consistent business behavior across all UI flows,
- clean and small Wt widgets that are easy to maintain,
- service APIs that are testable without Wt and without a DB,
- predictable error handling and user experience,
- and a scalable foundation for adding REST endpoints and background jobs without duplicating business logic.

16.3 Preventing logic leakage into the UI layer

In Wt applications, the UI layer is written in C++ and runs on the server. This can create a dangerous illusion: because the UI is “real code” (not browser JavaScript), it feels acceptable to place business logic directly inside widgets and signal handlers. Over time, this becomes one of the most expensive architectural failures: **logic leakage**.
Logic leakage happens when business decisions, policies, invariants, and persistence workflows spread into UI code. The UI layer becomes an untestable mixture of:

- widget wiring and layout,
- form parsing,
- domain rules,
- database calls,
- transaction control,
- security checks,
- and error translation.

This section explains the causes of leakage, the practical patterns to prevent it, and concrete Modern C++ designs that keep Wt widgets clean while the business logic remains authoritative in services and domain objects.

16.3.1 What “logic leakage” looks like in Wt

Leakage is not always obvious; it often starts as a harmless shortcut and slowly becomes the dominant style.

Typical leakage symptoms

- Widget callbacks contain complex “if” chains for business decisions.
- UI code calls repositories or ORM sessions directly.
- SQL queries are built inside Wt event handlers.
- Authorization rules are checked inconsistently across pages.
- The same validation logic appears in multiple widgets.
- UI code contains transaction boundaries (commit/rollback).
- UI code catches DB exceptions and decides business outcomes.

Why it is harmful

- **Inconsistency**: the same operation behaves differently depending on which page triggered it.
- **Bypass risk**: background jobs, admin screens, and APIs may skip UI-only checks.
- **Testing collapse**: verifying business behavior requires booting Wt, session state, and sometimes a DB.
- **Refactoring paralysis**: changing business rules requires editing many UI files, increasing regressions.
- **Security drift**: access checks scattered in UI become incomplete and vulnerable to oversight.

16.3.2 Root causes in server-side UI frameworks

In server-side frameworks like Wt, leakage happens for structural reasons:

- UI code is already C++ and can access everything, including repositories.
- Widget callbacks are natural places to "finish the job" quickly.
- Developers think "it is server-side, so it is safe to do logic here".
- Without explicit boundaries, dependencies become bidirectional: UI $\leftrightarrow$ DB.

The fix is not discipline alone. The fix is to enforce a design where leakage becomes **awkward** and the correct flow becomes **easy**.

16.3.3 The boundary rule

Adopt one rule that is simple enough to enforce:

> **Wt widgets may gather input and render output, but they must not implement business rules or call persistence directly.**

In practice, the UI may do:

- UX-only validation (empty field highlight),
- optimistic UI state updates (disable button, show spinner),
- formatting and presentation decisions.

But the UI must not do:

- eligibility decisions,
- state-transition rules,

- pricing or authorization policy,
- transaction orchestration,
- direct repository/SQL access.

16.3.4 The controller as the anti-leak barrier

Controllers prevent leakage by being the **only place** where UI events cross into the business logic world:

- UI emits an event → controller method called.
- Controller maps UI inputs to DTOs.
- Controller calls a service use-case.
- Controller maps stable results to UI-friendly outputs.
- UI renders only.

This makes it natural for the UI layer to remain thin.

16.3.5 A stable service API prevents UI decision-making

The fastest way to force business logic out of the UI is to ensure services return stable, structured results that are easy to render.

```
#include <string>
#include <utility>

enum class UseCaseError {
  InvalidInput,
  NotAuthorized,
```

```
  NotFound,
  Conflict,
  RuleViolation,
  TransientFailure,
  InternalError
};

template <class T>
struct Result {
  bool ok{false};
  T value{};
  UseCaseError error{UseCaseError::InternalError};
  std::string message{};

  static Result success(T v) {
    Result r; r.ok = true; r.value = std::move(v); return r;
  }
  static Result fail(UseCaseError e, std::string m) {
    Result r; r.ok = false; r.error = e; r.message = std::move(m); return r;
  }
};
```

With this pattern, the UI does not decide the meaning of failures; it only renders:

- `InvalidInput` → show field messages,
- `NotAuthorized` → show denied message,
- `Conflict` → show state conflict / refresh request,
- `TransientFailure` → show retry message.

16.3.6 Bad vs. good: a leakage example

Bad: business logic inside a widget callback

```
btn->clicked().connect([=] {
  // Leakage: repository access and business decisions inside UI
  auto email = emailEdit->text().toUTF8();
  auto pw    = pwEdit->text().toUTF8();

  if (email.empty() || pw.size() < 10) {
    status->setText("Invalid input");
    return;
  }

  if (repo.emailExists(email)) { // DB call in UI
    status->setText("Email already used");
    return;
  }

  if (!securityPolicy.mayRegister(currentUser)) { // policy in UI
    status->setText("Not allowed");
    return;
  }

  // transaction/commit in UI (very harmful)
  db.begin();
  repo.insertUser(email, hash(pw));
  audit.record("register");
  db.commit();

  status->setText("OK");
});
```

This code will grow until the widget becomes a hidden service layer.

Good: widget delegates to controller, controller delegates to service

Service contract

```
#include <cstdint>
#include <string>

struct RegisterUserRequest {
  std::string email;
  std::string password;
};

struct UserSummary {
  std::int64_t id{};
  std::string email;
};

struct IUserService {
  virtual ~IUserService() = default;
  virtual Result<UserSummary> register_user(const RegisterUserRequest& req) = 0;
};
```

Controller mapping

```
#include <Wt/WString.h>

class AccountController {
public:
  explicit AccountController(IUserService& svc) : users_(svc) {}

  Result<UserSummary> register_from_ui(const Wt::WString& email, const Wt::WString& pw) {
    RegisterUserRequest req;
    req.email = email.toUTF8();
    req.password = pw.toUTF8();
```

```
    // UI-side normalization only (trim, basic coercion)
    trim_in_place(req.email);

    return users_.register_user(req);
  }

private:
  static void trim_in_place(std::string& s) {
    auto is_space = [](unsigned char c){ return c==' '||c=='\t'||c=='\n'||c=='\r'; };
    while (!s.empty() && is_space(static_cast<unsigned char>(s.front())))
    ↪  s.erase(s.begin());
    while (!s.empty() && is_space(static_cast<unsigned char>(s.back())))  s.pop_back();
  }

  IUserService& users_;
};
```

Widget rendering

```
btn->clicked().connect([this] {
  auto r = ctrl_.register_from_ui(emailEdit_->text(), pwEdit_->text());

  if (!r.ok) {
    status_->setText(Wt::WString::fromUTF8(r.message));
    return;
  }

  status_->setText("Account created.");
});
```

Now:

- policy and transaction logic live inside the service,

- UI remains a presentation adapter,
- controller is a mapping/orchestration boundary.

16.3.7 Key techniques to prevent leakage

1) Disallow persistence dependencies in the UI layer

Enforce dependency direction:

$$\text{UI (Wt)} \rightarrow \text{Controllers} \rightarrow \text{Services} \rightarrow \text{Repositories/DB}$$

UI code should not even have access to repository headers in a clean build. Practically:

- separate targets/libraries (UI target does not link repository implementation),
- forward-declare service interfaces and inject them,
- keep persistence code in a separate module.

2) Use DTOs to cross boundaries

The UI should not pass domain entities directly. Use dedicated request/response structures:

- stable across refactors,
- minimal,
- independent from Wt types.

3) Make business rules "unskippable"

Put rules where they cannot be bypassed:

- invariants inside aggregates/entities,

- eligibility decisions inside policy/specification classes,
- state transitions in domain methods,
- authorization checks at the service boundary.

4) Centralize error mapping at the controller boundary

UI should not interpret exceptions. Services map internal failures to stable errors, controllers translate them into UI effects:

- show field errors,
- show global message,
- request refresh,
- redirect to login.

5) Treat the UI as replaceable

If you design the system so the UI could be replaced by:

- REST API,
- CLI tool,
- background scheduler,

without rewriting business rules, you have prevented leakage by design.

6) Keep widget callbacks extremely small

A practical metric:

> If a widget callback cannot fit on one screen, it probably contains leaked logic.

Callbacks should:

- read fields,
- call controller,
- apply result to UI.

16.3.8 Wt-specific leakage traps and how to avoid them

Trap A: session state mixed with domain state

Wt sessions and widget state should not become the source of truth for business state. Domain state belongs in the domain model and persistence, not in UI objects. Controllers may read session identity (user id, roles) but should not store business state in widgets.

Trap B: background operations triggered from UI

When UI triggers long workflows (payments, imports, report generation):

- controller calls a service to start the job,
- job is tracked in persistence or a queue,
- UI only polls status or receives updates.

Do not keep workflow state inside widget memory.

Trap C: UI tries to “fix” business errors

If a service returns `Conflict` (e.g., stale version), the UI should request refresh rather than attempting to merge state by guesswork. Conflict resolution rules belong to the domain/service level.

16.3.9 A practical checklist

- UI never includes repository headers and never opens transactions.
- UI calls controllers; controllers call services; services call repositories.
- Services return stable result types; UI never catches DB exceptions.
- Domain invariants are enforced in domain objects, not in widgets.
- Controllers perform mapping and flow control, not business decisions.
- Unit tests target domain + services; UI tests focus on rendering/wiring only.

16.3.10 Outcome

Preventing logic leakage keeps the Wt UI layer clean, predictable, and maintainable. It ensures that:

- business behavior is consistent across all entry points,
- security and policy enforcement remain centralized,
- tests can cover business logic without running Wt or a database,
- and UI development remains fast because widgets are not burdened by hidden service responsibilities.

Part VIII

Real-World Application Features

Advanced Forms

17.1 Professional validation techniques

Professional validation is a **multi-layer discipline** that protects correctness, security, and user experience at the same time. In Wt (a server-side C++ web toolkit), validation must be designed with a clear rule:

> **Client-side feedback improves usability, but server-side validation is the authority.**

Because all user input ultimately reaches your C++ server code, you must assume the client can be tampered with. Professional validation therefore combines:

- **Immediate feedback** while the user edits (client-side hints and visual cues),
- **Definitive validation** at the server before any state change,
- **Domain-level invariants** enforced in services/entities so they cannot be bypassed,
- **Security hardening** (length limits, canonicalization, encoding rules, rate limits).

17.1.1 Validation layers in a production-grade Wt application

Layer 1: UI-level constraints (usability)

These constraints exist to reduce friction and guide the user:

- required-field cues and placeholders,
- maximum length constraints,
- numeric range constraints for UI widgets,
- lightweight pattern checks (email-like, phone-like),
- immediate error highlighting (invalid style).

Important: UI-level validation must never be your only validation.

Layer 2: Controller/service input validation (authoritative)

Before calling persistence or changing domain state, validate:

- presence, type conversion, range checks,
- safe length bounds and resource limits,
- cross-field constraints (e.g., password confirmation),
- request-level rules (idempotency keys, replay limits).

Layer 3: Domain invariants (unskippable)

Your domain model (entities/value objects/policies) enforces:

- invariants and state transitions,

- business constraints (eligibility, caps),
- forbidden state combinations.

These rules must remain true regardless of whether the operation came from a Wt form, a REST endpoint, or a background job.

Layer 4: Persistence constraints (defense-in-depth)

Database constraints provide an additional safety net:

- unique constraints,
- foreign keys,
- check constraints where applicable,
- optimistic concurrency (version columns).

Persistence constraints are not a substitute for user-facing validation, but they prevent corruption when bugs slip in.

17.1.2 Wt fundamentals: validators, visual feedback, and authoritative checks

Wt provides a validator system that can be attached to form widgets. In general, validators can provide:

- **client-side feedback** (visual indication for invalid input),
- **server-side validation** when you call `validate()`.

When invalid, Wt typically marks the widget with an invalid style class (commonly `Wt-invalid`), allowing consistent styling for professional UI feedback.

17.1.3 Technique 1: Per-widget validators for simple forms

Use per-widget validators when:

- validation is local to one field,
- rules are simple (range, length, regex),
- cross-field dependencies are minimal.

```
#include <Wt/WLineEdit.h>
#include <Wt/WTextArea.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
#include <Wt/WIntValidator.h>
#include <Wt/WLengthValidator.h>
#include <Wt/WRegExpValidator.h>

auto age = addNew<Wt::WLineEdit>();
age->setPlaceholderText("Age");
age->setValidator(std::make_shared<Wt::WIntValidator>(0, 130));

auto username = addNew<Wt::WLineEdit>();
username->setPlaceholderText("Username");
// First apply a hard max length for resource safety:
username->setMaxLength(32);
// Then apply semantic constraints:
username->setValidator(std::make_shared<Wt::WRegExpValidator>("[A-Za-z0-9_]{3,32}"));

auto bio = addNew<Wt::WTextArea>();
bio->setPlaceholderText("Short bio");
bio->setValidator(std::make_shared<Wt::WLengthValidator>(0, 500));

auto status = addNew<Wt::WText>();
```

```
auto submit = addNew<Wt::WPushButton>("Save");

submit->clicked().connect([=] {
  // Authoritative check at submission time:
  const auto r1 = age->validate();
  const auto r2 = username->validate();
  const auto r3 = bio->validate();

  if (r1.state() != Wt::ValidationState::Valid ||
      r2.state() != Wt::ValidationState::Valid ||
      r3.state() != Wt::ValidationState::Valid) {
    status->setText("Please correct the highlighted fields.");
    return;
  }

  // Call controller/service here (not persistence directly).
  status->setText("Saved.");
});
```

Professional notes

- Use `setMaxLength()` as a **hard safety bound** to reduce memory and abuse risk.
- Use a validator for **semantic rules** (length range, pattern, integer range).
- Always validate again on submit, even if the UI shows real-time feedback.

17.1.4 Technique 2: Model-based validation with WFormModel and WTemplateFormView

For advanced forms, professional code usually moves validation into a **form model**:

- centralizes field definitions and validation rules,

- supports consistent error messages,
- enables cross-field validation cleanly,
- cleanly separates view (widgets) from validation and data handling.

A form model with field definitions and validators

```
#include <Wt/WFormModel.h>
#include <Wt/WString.h>
#include <Wt/WLengthValidator.h>
#include <Wt/WRegExpValidator.h>

class SignupModel : public Wt::WFormModel {
public:
  static const Field Email;
  static const Field Password;
  static const Field PasswordConfirm;

  SignupModel() {
    addField(Email);
    addField(Password);
    addField(PasswordConfirm);

    // Email: pragmatic pattern + length safety
    auto emailV = std::make_shared<Wt::WRegExpValidator>(".+@.+\\..+");
    emailV->setMandatory(true);
    setValidator(Email, emailV);

    auto pwLen = std::make_shared<Wt::WLengthValidator>(10, 128);
    pwLen->setMandatory(true);
    setValidator(Password, pwLen);

    auto pwcLen = std::make_shared<Wt::WLengthValidator>(10, 128);
```

```
    pwcLen->setMandatory(true);
    setValidator(PasswordConfirm, pwcLen);
  }

  bool validate() override {
    // First, run per-field validation.
    bool ok = Wt::WFormModel::validate();

    // Cross-field rule: passwords must match.
    const auto pw  = valueText(Password).toUTF8();
    const auto pwc = valueText(PasswordConfirm).toUTF8();

    if (pw != pwc) {
      setValidation(PasswordConfirm,
        Wt::WValidator::Result(Wt::ValidationState::Invalid,
          Wt::WString("Passwords do not match.")));
      ok = false;
    }

    // Example canonicalization rule: normalize/trim email (view-independent).
    auto email = valueText(Email).toUTF8();
    trim_in_place(email);
    setValueText(Email, Wt::WString::fromUTF8(email));

    return ok;
  }

private:
  static void trim_in_place(std::string& s) {
    auto is_space = [](unsigned char c){ return c==' '||c=='\t'||c=='\n'||c=='\r'; };
    while (!s.empty() && is_space(static_cast<unsigned char>(s.front())))
    ↪  s.erase(s.begin());
    while (!s.empty() && is_space(static_cast<unsigned char>(s.back())))  s.pop_back();
```

```
  }
};

const Wt::WFormModel::Field SignupModel::Email("email");
const Wt::WFormModel::Field SignupModel::Password("password");
const Wt::WFormModel::Field SignupModel::PasswordConfirm("passwordConfirm");
```

Binding the model to a template-based view

```
#include <Wt/WTemplateFormView.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class SignupView : public Wt::WTemplateFormView {
public:
  explicit SignupView(std::shared_ptr<SignupModel> model)
    : model_(std::move(model)) {

    setModel(model_);

    // Editors:
    email_ = bindWidget(SignupModel::Email, std::make_unique<Wt::WLineEdit>());
    pw_    = bindWidget(SignupModel::Password, std::make_unique<Wt::WLineEdit>());
    pwc_   = bindWidget(SignupModel::PasswordConfirm, std::make_unique<Wt::WLineEdit>());

    pw_->setEchoMode(Wt::EchoMode::Password);
    pwc_->setEchoMode(Wt::EchoMode::Password);

    // Save button:
    auto save = bindWidget("save", std::make_unique<Wt::WPushButton>("Create account"));
    status_ = bindWidget("status", std::make_unique<Wt::WText>());
```

```
    save->clicked().connect([this] { onSubmit(); });

    // Optional: validate on blur / change to provide earlier feedback.
    email_->changed().connect([this] { updateModelField(SignupModel::Email);
    ↪  validateField(SignupModel::Email); });
    pw_->changed().connect([this]    { updateModelField(SignupModel::Password); });
    pwc_->changed().connect([this]   { updateModelField(SignupModel::PasswordConfirm); });
  }

private:
  void onSubmit() {
    // Pull UI values into the model:
    updateModel();

    // Authoritative validation:
    if (!model_->validate()) {
      // Update the view to show validation messages:
      updateView(model_.get());
      status_->setText("Please fix the highlighted fields.");
      return;
    }

    // Call controller/service here (not inside the view).
    status_->setText("Account request accepted.");
  }

  std::shared_ptr<SignupModel> model_;
  Wt::WLineEdit* email_{};
  Wt::WLineEdit* pw_{};
  Wt::WLineEdit* pwc_{};
  Wt::WText* status_{};
};
```

Why model-based validation is professional

- Rules live in one place and can be unit-tested without Wt widgets.
- Views remain mostly presentation and binding.
- Cross-field rules are first-class, not hacks in callbacks.

17.1.5 Technique 3: Cross-field and conditional validation (real-world rules)

Professional forms often contain conditional rules:

- “If country is X, postal code must match pattern Y.”
- “If user selects business account, VAT number becomes mandatory.”
- “If checkbox enabled, additional fields must be provided.”

Implement these rules in:

- the form model `validate()` override (preferred for UI form rules),
- or the service layer (preferred when rules are business policies).

```
// Example snippet inside a WFormModel::validate() override:
if (valueText(Field("accountType")) == "business") {
  auto vat = valueText(Field("vat")).toUTF8();
  if (vat.empty()) {
    setValidation(Field("vat"),
      Wt::WValidator::Result(Wt::ValidationState::Invalid, "VAT number is required."));
    ok = false;
  }
}
```

17.1.6 Technique 4: Server-side uniqueness and anti-race validation

Some validation depends on server state:

- username/email uniqueness,
- invitation token validity,
- quota availability,
- permission checks.

Professional rule: treat these as **service-layer validations**, and enforce them with:

- transaction boundaries,
- and database constraints (e.g., unique index) as final defense.

A robust flow is:

1. UI checks syntax locally (optional).
2. Service checks uniqueness under transaction.
3. DB unique constraint guarantees correctness even under concurrency.
4. Service maps conflict to a stable error (e.g., `Conflict`).

17.1.7 Technique 5: Validation timing (when to validate)

Validate on submit (mandatory)

Always validate on submit before any persistent change.

Validate on blur/change (recommended)

For a professional UX:

- validate after the field is edited and loses focus,
- avoid aggressive validation on every keystroke for expensive rules,
- validate cross-field rules when dependent values change.

Debounced server checks (advanced)

If you perform server checks (e.g., “is username available”), debounce requests:

- wait for a pause in typing,
- cancel older checks,
- and never block submission on a stale check: submission revalidates anyway.

17.1.8 Technique 6: Canonicalization and safe bounds (security-grade validation)

Professional validation is not only about correctness, it is also about safety.

Hard length limits

Set maximum lengths at the UI and re-check on the server:

- prevents oversized payload abuse,
- reduces memory pressure,
- limits log and error amplification.

Canonicalization (normalize before validate)

Examples:

- trim leading/trailing whitespace,
- normalize line endings in text areas,
- apply consistent case handling where appropriate (e.g., emails often treated case-insensitively for domain parts),
- reject or normalize forbidden control characters.

Never trust regex alone for complex formats

For email, URL, and international phone numbers:

- use pragmatic checks for UX,
- and rely on service-layer confirmation workflows where needed (email verification) rather than impossible “perfect regex”.

Output encoding is not input validation

Input validation reduces risk, but you must also:

- encode output correctly for HTML contexts,
- avoid injecting raw user text into HTML/JS,
- and keep templates safe.

17.1.9 Technique 7: Professional error reporting (messages and UX)

Field-level messages

Show precise messages near fields:

- what is wrong,
- what format is expected,
- and how to fix it.

Form-level summary

Provide a summary at the top for accessibility and faster correction:

- “3 fields need attention” and list them,
- focus the first invalid field.

Stable error categories from services

When a failure comes from business logic or concurrency, use stable categories:

- invalid input vs conflict vs not authorized,
- and map each to predictable UI behavior.

17.1.10 Technique 8: Testing validation professionally

A production system validates behavior with:

- unit tests for form models (cross-field logic),
- unit tests for service input validation and policy enforcement,

- integration tests for DB constraints (uniqueness, foreign keys),
- limited UI tests to confirm wiring and rendering, not business rules.

```
#include <cassert>
#include <memory>

static void test_password_confirmation_rule() {
  auto m = std::make_shared<SignupModel>();
  m->setValueText(SignupModel::Email, "a@b.com");
  m->setValueText(SignupModel::Password, "1234567890");
  m->setValueText(SignupModel::PasswordConfirm, "DIFFERENT");

  bool ok = m->validate();
  assert(!ok);
  assert(m->validation(SignupModel::PasswordConfirm).state() !=
  ↪  Wt::ValidationState::Valid);
}
```

17.1.11 Checklist: what “professional validation” means in Chapter 17

- UI validation exists for usability, not authority.
- Server-side validation is always executed before state changes.
- Model-based validation is preferred for advanced multi-field forms.
- Cross-field rules are centralized (model/service), not duplicated in widgets.
- Server-state rules (uniqueness, quotas, permissions) belong to services and are protected by DB constraints.
- Hard limits (`setMaxLength()`, size bounds) are applied defensively.

- Inputs are canonicalized before validation and use-case execution.
- Validation messages are precise, consistent, and accessible.
- Tests target models and services; UI tests remain minimal.

17.1.12 Outcome

When validation is implemented professionally in Wt advanced forms, you gain:

- stronger security against tampering and abuse,
- consistent business correctness under concurrency,
- improved UX through immediate and clear feedback,
- maintainable form logic through model-based validation,
- and a testable system where validation can be verified without running the full UI.

17.2 Error messaging

Error messaging is a critical part of professional form handling. It is the primary communication channel between the systems internal validation, business logic, and the end user. Poor error messaging causes confusion, user frustration, security leaks, and inconsistent behavior across the application. High-quality error messaging, by contrast, improves usability, trust, and correctness while preserving system integrity.
In Wt-based server-side applications, error messaging must be designed with a clear separation between:

- user-facing feedback,
- service-level outcomes,

- domain rule enforcement,
- and internal diagnostic information.

17.2.1 Goals of professional error messaging

A professional error messaging system must achieve the following goals simultaneously:

- Clearly explain what went wrong from the users perspective.
- Indicate how the user can fix the issue, when applicable.
- Avoid exposing internal implementation details or security-sensitive information.
- Remain consistent across all forms and interaction flows.
- Support localization and accessibility.
- Be testable and deterministic.

17.2.2 Error categories and their meaning

Error messages must reflect the origin of the failure. Mixing categories leads to misleading UI behavior.

Input validation errors

These errors arise from malformed or incomplete user input:

- required fields left empty,
- invalid formats,
- values outside permitted ranges,

- mismatched confirmation fields.

These errors should be attached directly to the affected fields.

Business rule violations

These errors occur when syntactically valid input violates domain rules:

- account already exists,
- password does not meet policy,
- order state does not allow the requested action,
- quota or limit exceeded.

Such errors should be explained in domain language, not technical terms.

Authorization and permission errors

These errors indicate that the user is not allowed to perform the action:

- insufficient privileges,
- expired session,
- restricted operation.

Messages must be firm but neutral, without revealing sensitive policy details.

Conflicts and concurrency errors

These occur when the system state has changed:

- duplicate submission,

- optimistic locking failure,
- stale form data.

The user should be instructed to refresh or retry.

System and transient failures

These errors originate outside business logic:

- database timeouts,
- network failures,
- external service outages.

Messages must remain generic and never expose internal diagnostics.

17.2.3 Field-level error messaging in Wt

Wt supports attaching validators directly to form widgets. Field-level messages provide immediate, localized feedback.

```
auto email = addNew<Wt::WLineEdit>();
email->setPlaceholderText("Email address");
email->setMaxLength(254);

auto emailValidator =
  std::make_shared<Wt::WRegExpValidator>(".+@.+\\..+");
emailValidator->setMandatory(true);
emailValidator->setInvalidText("Please enter a valid email address.");
email->setValidator(emailValidator);
```

At submission time, validation must always be rechecked:

```
if (email->validate().state() != Wt::ValidationState::Valid) {
  status_->setText("Please correct the highlighted fields.");
  return;
}
```

17.2.4 Form-level error summaries

Complex forms benefit from a global summary message that complements field-level feedback.

```
if (!model_->validate()) {
  updateView(model_.get());
  summary_->setText("The form contains errors. Please review and fix them.");
  return;
}
```

Form-level summaries improve accessibility and reduce user frustration in large forms.

17.2.5 Stable service error categories

Services should never return UI strings directly. Instead, they return structured outcomes with stable error categories.

```
enum class UseCaseError {
  InvalidInput,
  NotAuthorized,
  NotFound,
  Conflict,
  RuleViolation,
  TransientFailure,
  InternalError
};

template <typename T>
```

```
struct Result {
  bool ok{false};
  T value{};
  UseCaseError error{UseCaseError::InternalError};
  std::string message{};

  static Result success(T v) {
    Result r; r.ok = true; r.value = std::move(v); return r;
  }

  static Result fail(UseCaseError e, std::string m) {
    Result r; r.ok = false; r.error = e; r.message = std::move(m); return r;
  }
};
```

17.2.6 Controller-level error mapping

Controllers translate service outcomes into user-facing messages without guessing intent.

```
auto r = userService_.register_user(req);

if (!r.ok) {
  switch (r.error) {
    case UseCaseError::InvalidInput:
      status_->setText("Some fields contain invalid values.");
      break;
    case UseCaseError::Conflict:
      status_->setText("An account with this email already exists.");
      break;
    case UseCaseError::NotAuthorized:
      status_->setText("You are not authorized to perform this action.");
      break;
    case UseCaseError::TransientFailure:
```

```
      status_->setText("Temporary system issue. Please try again.");
      break;
    default:
      status_->setText("An unexpected error occurred.");
      break;
  }
  return;
}
```

This ensures consistency across all entry points.

17.2.7 Security principles in error messaging

Professional systems follow strict security rules:

- Never reveal whether a specific account exists.
- Never expose stack traces, SQL errors, or exception messages.
- Use identical messages for authentication failures.
- Log detailed diagnostics internally, not in the UI.

```
// Correct: prevents account enumeration
status_->setText("Invalid credentials.");
```

17.2.8 Internationalization-ready error messages

Error messages must be translatable. UI code should reference message keys, not hardcoded text.

```
status_->setText(tr("error.email.invalid"));
```

This enables localization without recompilation and keeps messaging consistent.

17.2.9 Visual and accessibility considerations

Professional error messaging includes visual clarity:

- consistent styling for invalid fields,
- clear contrast for error text,
- focus on the first invalid field,
- compatibility with assistive technologies.

17.2.10 Testing error messaging

Error messaging must be testable at multiple levels:

- unit tests for form models,
- service tests for error category mapping,
- controller tests for message translation,
- UI tests for correct placement and visibility.

```
static void test_duplicate_email_message() {
  auto r = Result<int>::fail(UseCaseError::Conflict, "");
  assert(r.error == UseCaseError::Conflict);
}
```

17.2.11 Checklist for professional error messaging

- Field-level errors are precise and localized.
- Form-level summaries guide correction.

- Services return stable error categories.
- Controllers map errors consistently.
- UI never interprets infrastructure failures.
- Messages are secure, neutral, and actionable.
- Localization is supported by design.
- Error behavior is covered by tests.

17.2.12 Outcome

Well-designed error messaging transforms validation and failure handling from a source of confusion into a structured, predictable, and trustworthy user experience. It reinforces system correctness, protects security boundaries, and keeps advanced Wt applications maintainable as they grow.

17.3 Server-side UX considerations

In Wt, the UI is rendered and controlled from the server, and user interaction flows are processed through server-side event handling. This architecture enables strong consistency and security, but it also introduces UX challenges that do not exist (or are less visible) in purely client-side frameworks. A professional Wt application must treat UX as an engineering discipline: latency, concurrency, feedback timing, state synchronization, error recovery, and accessibility must be designed deliberately.
This section covers server-side UX considerations for advanced forms in production-grade Wt applications, focusing on patterns that preserve responsiveness and clarity while keeping the business logic authoritative.

17.3.1 The server-side UX constraint model

Server-side UI means:

- user actions trigger signals that result in server code execution,
- UI updates are shipped back to the browser,
- network latency and server scheduling are part of the perceived “UI speed”,
- each session carries state that must remain coherent under multi-step form workflows.

Therefore, professional UX design in Wt forms must address:

- perceived responsiveness,
- feedback clarity,
- safe interaction under retries and duplicates,
- navigation predictability,
- and correctness under concurrent edits.

17.3.2 Principle 1: Optimize perceived responsiveness

Immediate feedback on interaction

On every meaningful user action (submit, add row, apply coupon), do not wait silently for the server round-trip. Provide immediate UI feedback:

- disable the submit button to prevent double submissions,
- show a status text (“Processing...”), or a progress indicator,

- visually indicate which section is being processed.

```
submit_->clicked().connect([this] {
  submit_->disable();
  status_->setText("Processing...");

  // Call controller/service (server-side operation):
  auto r = controller_.submit_form(collect_request());

  if (!r.ok) {
    status_->setText(Wt::WString::fromUTF8(r.message));
    submit_->enable();
    return;
  }

  status_->setText("Saved.");
});
```

Avoid expensive work in the UI thread

In server-side UI, heavy operations block the sessions responsiveness. For expensive validation or business operations:

- separate the operation into a background job,
- show a status view,
- poll for completion or push updates when available.

17.3.3 Principle 2: Prevent duplicate submissions and multi-click hazards

Duplicate submissions are more common in server-side workflows because:

- network latency encourages re-clicking,

- users may use back/forward navigation,
- browser retries may occur.

Professional approaches:

- disable submit controls after click,
- add idempotency tokens to critical operations,
- enforce idempotency in services (not only in the UI).

```
struct SubmitRequest {
  std::string idempotencyKey;
  std::string payload;
};
```

17.3.4 Principle 3: Validate progressively, but confirm authoritatively

A professional form experience uses multiple validation moments:

- early field-level feedback (format, length, basic constraints),
- cross-field feedback when dependent fields change,
- authoritative validation at submission time.

The UX goal is to reduce surprise at submit time, but the security goal is to ensure the server remains the authority.

```
email_->changed().connect([this] {
  // Lightweight feedback only:
  if (email_->validate().state() != Wt::ValidationState::Valid) {
    hint_->setText("Check the email format.");
  } else {
    hint_->setText("");
  }
});
```

17.3.5 Principle 4: Use clear error placement and recovery paths

Error messaging must support quick recovery:

- field-level errors near the field,
- a form-level summary for large forms,
- focus the first invalid field,
- preserve user-entered values where safe.

Preserving values is especially important in server-side forms:

- avoid losing user input on partial failures,
- avoid resetting the entire form on one invalid field.

```
if (!model_->validate()) {
  updateView(model_.get());
  summary_->setText("Please fix the highlighted fields.");
  focusFirstInvalid();
  return;
}
```

17.3.6 Principle 5: Keep long forms stable and navigable

Advanced forms often span many sections. Server-side UX must reduce cognitive load:

- break large forms into steps (wizard flow),
- save draft state (server-side or persistence),
- provide “Back” without losing state,
- show progress indicators.

Wizard-style flow (multi-step forms)

In a multi-step form:

- validate the current step only,
- store intermediate state safely (session or draft record),
- allow users to revise previous steps predictably.

```
enum class Step { Profile, Address, Review };

void goToStep(Step s) {
  step_ = s;
  renderStep();
}
```

17.3.7 Principle 6: Design for session state and refresh safety

Server-side UI sessions introduce UX edge cases:

- user refreshes the page,
- session expires mid-form,
- user opens the same form in multiple tabs,
- server restarts invalidate sessions.

Professional UX handling includes:

- clear session-expired messages,
- safe redirection to login,
- draft persistence for critical long forms,
- conflict detection when multiple tabs edit the same resource.

17.3.8 Principle 7: Concurrency and conflict UX

For forms editing shared data, conflicts happen:

- another user updates the record while you edit,
- or the same user edits in two tabs.

Professional solutions:

- optimistic concurrency with version fields,
- conflict responses mapped to a user-friendly message,
- “Reload and retry” workflows,
- optional merge UI for advanced cases.

```
if (result.error == UseCaseError::Conflict) {
  status_->setText("This data changed on the server. Please reload and try again.");
  reloadButton_->show();
  return;
}
```

17.3.9 Principle 8: Accessibility and keyboard-first flow

Professional UX includes accessibility, even for server-side rendering:

- keyboard navigation order must be correct,
- focus must move predictably after errors and submissions,
- form errors should be readable and discoverable,
- widgets must have meaningful labels.

Focus management

After validation failure:

- focus the first invalid widget,
- ensure error is visible near it,
- avoid forcing full-page jumps without context.

```
void focusFirstInvalid() {
  if (email_->validate().state() != Wt::ValidationState::Valid) { email_->setFocus();
  ↪  return; }
  if (name_->validate().state()  != Wt::ValidationState::Valid) { name_->setFocus();
  ↪  return; }
  if (age_->validate().state()   != Wt::ValidationState::Valid) { age_->setFocus();
  ↪  return; }
}
```

17.3.10 Principle 9: Data binding and safe re-rendering

Server-side UI updates must avoid surprising resets:

- re-render only the necessary part of the form,
- preserve user-entered values,
- avoid tearing down widgets on minor updates.

When using model-based forms:

- update the model from UI (`updateModel()`),
- validate,
- update only the relevant view parts (`updateView()`),
- maintain stable widget identity where possible.

17.3.11 Principle 10: Professional messaging and status design

Server-side UX requires disciplined messaging:

- status messages must be short and actionable,
- do not flood the UI with logs or technical details,
- separate warnings (non-blocking) from errors (blocking),
- ensure success messages confirm the action and next step.

```
status_->setText("Saved successfully. You may continue editing or return to the dashboard.");
```

17.3.12 A practical UX checklist for Wt advanced forms

- Disable submit and show progress feedback to prevent duplicates.
- Use progressive validation but always validate on submit server-side.
- Preserve inputs on errors and focus the first invalid field.
- Provide form summaries for long forms and clear navigation steps.
- Handle session expiry with clear recovery (login redirect, draft restore).
- Use optimistic concurrency and provide conflict recovery UI.
- Avoid heavy work in UI callbacks; move it to services or background jobs.
- Keep UI updates minimal and stable; avoid unnecessary widget rebuilds.
- Ensure accessibility: labels, focus control, predictable tab order.
- Keep messages consistent, actionable, and non-technical.

17.3.13 Outcome

When server-side UX is engineered properly in Wt, advanced forms achieve:

- responsiveness despite network round-trips,
- clear feedback under validation and business rule enforcement,
- safe behavior under duplicate submissions and concurrency,
- predictable navigation in multi-step workflows,
- and maintainable UI code because business logic stays in controllers and services.

Data Tables and Views

18.1 Tables

Tables are one of the most common and most demanding UI components in production web applications. They are not merely “visual lists”; they are the main interaction surface for browsing, filtering, sorting, and acting on large datasets. In Wt, tables are implemented using server-side models and views. This provides strong consistency and security, but it requires disciplined design to keep performance, responsiveness, and correctness under real-world constraints.

This section establishes professional patterns for building scalable, maintainable, and testable table views in Wt, including model/view separation, pagination, sorting, filtering, selection, actions, and safe rendering.

18.1.1 What makes tables a production feature?

A table becomes a production-grade feature when it supports:

- large datasets (thousands to millions of records),
- paging and incremental loading,
- server-side sorting and filtering,

- role-based action availability,
- stable formatting and safe rendering,
- responsive UX under latency,
- consistent error handling and recoverable failures.

If a table is built as a simple in-memory container of all rows, it will fail under real load.

18.1.2 The Wt model/view approach

Wt table rendering follows a model/view pattern:

- **Models** provide data and structure (rows, columns, values).
- **Views** render that data (table widgets, columns, cell widgets).

This separation is essential for:

- reuse (same model can back different views),
- performance (views can render only visible parts),
- testability (model logic can be tested without UI),
- clean layering (table views should not embed persistence).

18.1.3 Choosing the right table view approach

Wt offers multiple ways to present tabular data. The professional choice depends on dataset size and interactivity requirements.

- **Small datasets (few hundred rows)**: in-memory models are acceptable.

- **Medium datasets (thousands)**: prefer lazy loading and server-side paging.
- **Large datasets (tens of thousands+)**: require server-side paging, indexed sorting/filtering, and optimized queries.

Professional design assumes that data grows.

18.1.4 Data table requirements: structure and responsibilities

Table view responsibilities

A table view should:

- render column headers and row values,
- provide selection and row actions,
- handle user-triggered sorting and filtering UI,
- display loading and error states.

It should **not**:

- call repositories directly,
- execute SQL,
- implement business rules.

Controller/service responsibilities

Controllers and services should:

- interpret filter and sort requests,

- enforce authorization for row actions,
- fetch page data,
- map results into DTOs or view models,
- enforce stable error categories.

18.1.5 A professional table pipeline

A scalable table pipeline is:

1. UI captures paging/sort/filter parameters.
2. Controller validates and normalizes the request.
3. Service performs query orchestration, business constraints, authorization.
4. Repository executes optimized DB queries with limit/offset or keyset paging.
5. Service returns a page result: total count (optional) + rows.
6. UI updates the table model/view.

18.1.6 Defining a row DTO for tables

Tables should render stable DTOs rather than domain entities.

```
#include <cstdint>
#include <string>

struct UserRow {
  std::int64_t id{};
  std::string email;
```

```
  std::string displayName;
  std::string status; // e.g., "Active", "Locked"
};
```

18.1.7 Defining a paged result contract

Professional tables require paging. Define a generic result.

```
#include <vector>
#include <cstdint>

template <class Row>
struct PageResult {
  std::vector<Row> rows;
  std::int64_t totalRows{-1}; // -1 if not computed
  int pageIndex{0};
  int pageSize{50};
};
```

18.1.8 Table request parameters: sort, filter, paging

```
#include <string>
#include <optional>

enum class SortDir { Asc, Desc };

struct SortSpec {
  std::string column; // must be validated against allowed columns
  SortDir dir{SortDir::Asc};
};

struct TableQuery {
  int pageIndex{0};
```

```
  int pageSize{50};
  std::optional<SortSpec> sort;
  std::string search; // a global search term (optional)
};
```

18.1.9 Service interface for table data

Keep UI independent from persistence by exposing a service method:

```
struct IUserTableService {
  virtual ~IUserTableService() = default;
  virtual PageResult<UserRow> list_users(const TableQuery& q) = 0;
};
```

18.1.10 Implementing a Wt table model

A robust approach is to implement a custom model that holds the current page.

```
#include <Wt/WAbstractTableModel.h>
#include <Wt/WString.h>
#include <vector>

class UserTableModel final : public Wt::WAbstractTableModel {
public:
  enum Col { Id, Email, DisplayName, Status, ColCount };

  void setPage(PageResult<UserRow> page) {
    beginReset();
    page_ = std::move(page);
    endReset();
  }

  int rowCount(const Wt::WModelIndex& parent = Wt::WModelIndex()) const override {
```

```
  (void)parent;
  return static_cast<int>(page_.rows.size());
}

int columnCount(const Wt::WModelIndex& parent = Wt::WModelIndex()) const override {
  (void)parent;
  return static_cast<int>(ColCount);
}

Wt::cpp17::any data(const Wt::WModelIndex& index,
                    Wt::ItemDataRole role = Wt::ItemDataRole::Display) const override {
  if (!index.isValid() || role != Wt::ItemDataRole::Display) return {};

  const auto& r = page_.rows.at(static_cast<std::size_t>(index.row()));

  switch (static_cast<Col>(index.column())) {
    case Id:          return Wt::WString::fromUTF8(std::to_string(r.id));
    case Email:       return Wt::WString::fromUTF8(r.email);
    case DisplayName: return Wt::WString::fromUTF8(r.displayName);
    case Status:      return Wt::WString::fromUTF8(r.status);
    default:          return {};
  }
}

Wt::cpp17::any headerData(int section,
                          Wt::Orientation orientation,
                          Wt::ItemDataRole role = Wt::ItemDataRole::Display) const override {
  if (orientation != Wt::Orientation::Horizontal || role != Wt::ItemDataRole::Display)
  ↪  return {};

  switch (static_cast<Col>(section)) {
    case Id:          return Wt::WString("ID");
    case Email:       return Wt::WString("Email");
```

```
      case DisplayName: return Wt::WString("Name");
      case Status:      return Wt::WString("Status");
      default:          return {};
    }
  }

private:
  PageResult<UserRow> page_{};
};
```

18.1.11 Rendering a view with sorting and paging controls

In production, the table is accompanied by controls:

- search box,
- page navigation,
- page size selection,
- sort toggles.

```
#include <Wt/WContainerWidget.h>
#include <Wt/WTableView.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class UsersTableView final : public Wt::WContainerWidget {
public:
  UsersTableView(IUserTableService& svc)
    : svc_(svc) {

    search_ = addNew<Wt::WLineEdit>();
```

```
    search_->setPlaceholderText("Search...");

    auto searchBtn = addNew<Wt::WPushButton>("Search");
    status_ = addNew<Wt::WText>();

    model_ = std::make_shared<UserTableModel>();

    table_ = addNew<Wt::WTableView>();
    table_->setModel(model_);
    table_->setSortingEnabled(false); // We will implement server-side sorting explicitly.

    // Column sizing hints (keep it stable for UX):
    table_->setColumnWidth(UserTableModel::Id, 70);
    table_->setColumnWidth(UserTableModel::Email, 260);
    table_->setColumnWidth(UserTableModel::DisplayName, 220);
    table_->setColumnWidth(UserTableModel::Status, 120);

    searchBtn->clicked().connect([this] {
      query_.pageIndex = 0;
      query_.search = search_->text().toUTF8();
      reload();
    });

    reload();
  }

private:
  void reload() {
    status_->setText("Loading...");
    auto page = svc_.list_users(query_);
    model_->setPage(std::move(page));
    status_->setText("");
  }
```

```
  IUserTableService& svc_;
  std::shared_ptr<UserTableModel> model_;
  Wt::WTableView* table_{};
  Wt::WLineEdit* search_{};
  Wt::WText* status_{};
  TableQuery query_{};
};
```

18.1.12 Server-side sorting: safe and validated

Sorting must be validated. Never accept raw column names from the UI and inject them into SQL. The service should map allowed sort columns to known fields.

```
static bool is_allowed_sort_column(const std::string& c) {
  return c == "email" || c == "name" || c == "status" || c == "id";
}
```

18.1.13 Filtering and searching: predictable behavior

Search UX must be predictable:

- global search should clearly define what it searches (email, name, etc.),
- filtering should not silently change sorting,
- filters should be visible and resettable.

Professional filtering considerations:

- apply debouncing for search updates (avoid flooding server with requests),
- keep filters in the URL or session state for navigation persistence,
- ensure filters are authorized (some users may not see certain rows).

18.1.14 Pagination strategy: offset vs. keyset

For very large datasets, offset paging can become slow. Production systems often use:

- offset paging for small/medium tables,
- keyset paging (seek method) for large tables where users navigate forward/back predictably.

The service layer should abstract the paging strategy so the UI does not depend on DB implementation.

18.1.15 Row actions: selection and commands

Professional tables include row actions:

- open details,
- edit,
- activate/deactivate,
- delete (rare, usually soft-delete),
- export.

Row actions must be:

- authorized server-side,
- confirmed in the UI where destructive,
- safe under concurrency (conflict handling),
- idempotent where possible.

18.1.16 Rendering safety: escaping and output discipline

Table cells often include user-provided text. Professional systems ensure:

- output is safely rendered and not interpreted as HTML unless intended,
- formatting rules are consistent across all views,
- dangerous content is never injected into templates without escaping.

18.1.17 Performance and UX considerations

Production-grade table performance depends on:

- keeping queries indexed and bounded,
- loading only needed columns,
- using paging rather than full loads,
- avoiding per-row expensive operations in rendering,
- caching stable reference data where appropriate.

UX considerations include:

- show loading state when reloading pages,
- preserve selection when possible,
- provide clear empty state messaging (no results),
- provide consistent column widths and alignment.

18.1.18 Testing strategy

A professional table feature is tested at multiple layers:

- service tests: sort/filter normalization and authorization,
- repository integration tests: query correctness and performance constraints,
- UI tests: basic rendering, loading state, empty state, action availability.

18.1.19 Outcome

A table designed with these principles becomes a stable production feature:

- scalable to large datasets,
- consistent under sorting/filtering/paging,
- secure in input handling and output rendering,
- responsive in server-side UX,
- and maintainable because business logic stays in services and controllers.

18.2 Pagination

Pagination is not a cosmetic feature. In production web applications, it is a fundamental scalability mechanism that protects the server, the database, and the user experience from unbounded dataset growth. In Wt, pagination must be engineered with server-side constraints in mind: network round-trips, session state, model/view rendering cost, and database query efficiency all directly affect perceived responsiveness.

This section presents professional pagination design for Wt data tables and views, including paging models, offset vs. keyset pagination, server-side sorting constraints, stable navigation, caching, and UX patterns that keep the table responsive under real-world load.

18.2.1 What pagination must guarantee

A production-grade pagination system must guarantee:

- **bounded work**: each page request has strict CPU, memory, and query bounds.
- **stable ordering**: page navigation uses a consistent sort order to avoid duplicates/missing rows.
- **predictable UX**: users understand where they are and can move forward/back reliably.
- **safe concurrency behavior**: changes during paging do not corrupt the users view.
- **security correctness**: authorization filters apply to paging queries.

Pagination is always **server-authoritative**. The UI only requests a page; the service decides what rows are visible and how paging is implemented.

18.2.2 Core pagination parameters

At minimum, paging requires:

- `pageIndex`: zero-based page number (or 1-based, but be consistent),
- `pageSize`: number of rows per page (bounded),
- `sort`: stable ordering specification,
- `filter/search`: optional filtering constraints.

```
#include <string>
#include <optional>

enum class SortDir { Asc, Desc };

struct SortSpec {
  std::string column;
  SortDir dir{SortDir::Asc};
};

struct TableQuery {
  int pageIndex{0};
  int pageSize{50};
  std::optional<SortSpec> sort;
  std::string search;
};
```

18.2.3 Page size discipline and safety

Never allow unbounded `pageSize`. Enforce hard bounds:

- default size for typical use (e.g., 25–50),
- maximum size to protect resources (e.g., 100–200),
- optionally a minimum size for UI consistency.

```
static int clamp_page_size(int requested) {
  constexpr int kMin = 10;
  constexpr int kMax = 200;
  if (requested < kMin) return kMin;
  if (requested > kMax) return kMax;
  return requested;
}
```

This is not just UX; it is a resource control policy.

18.2.4 Stable ordering: the non-negotiable rule

Paging is meaningless without stable ordering. If the query order is not stable, the user will see:

- duplicates across pages,
- missing rows,
- random reorder when data changes.

Therefore:

- always apply a deterministic sort order,
- if sorting by a non-unique column, append a unique tie-breaker (e.g., `id`).

```
struct NormalizedSort {
  std::string primary; // validated
  SortDir dir{SortDir::Asc};
  bool tieBreakWithId{true};
};
```

18.2.5 Pagination strategies: offset vs. keyset

Offset pagination

Offset pagination uses `LIMIT` and `OFFSET` (or equivalent):

- **Pros**: simple UI (page numbers), easy to jump to arbitrary pages.
- **Cons**: can become slow for deep pages because the database must scan/skip many rows.

- **Concurrency**: inserts/deletes can shift offsets, causing duplicates/missing rows across navigation.

Offset pagination works well for:

- small to medium datasets,
- admin screens where jump-to-page is valuable,
- analytics tables where approximate browsing is acceptable.

Keyset (seek) pagination

Keyset pagination uses a cursor derived from the last row:

- **Pros**: fast for deep navigation; uses indexes efficiently.
- **Cons**: page numbers are not natural; jumping to page 200 is not trivial.
- **Concurrency**: more stable under inserts/deletes when using stable ordering.

Keyset is recommended for:

- very large datasets,
- “infinite scroll” UX,
- time-ordered feeds,
- high-traffic tables where performance is critical.

18.2.6 A unified service contract for paging

The UI should not depend on SQL strategy. Your service contract returns a page result.

```
#include <vector>
#include <cstdint>

template <class Row>
struct PageResult {
  std::vector<Row> rows;
  std::int64_t totalRows{-1}; // -1 when not computed or not needed
  int pageIndex{0};
  int pageSize{50};
  bool hasNext{false};
  bool hasPrev{false};
};
```

18.2.7 Computing total count: when to do it

Counting total rows can be expensive for large filtered datasets. Professional design treats `totalRows` as optional:

- compute it when the UI truly needs “Page X of Y”,
- skip it when the UI only needs “Next” and “Previous”,
- cache it briefly if it is expensive and repeated.

A common practice:

- compute `totalRows` on the first page load,
- avoid recomputing on every page unless filters change.

18.2.8 Implementing pagination in a Wt table model

A Wt table view should not attempt to load all rows. Instead:

- keep only the current page in memory,
- reload the model when the page changes.

```
#include <Wt/WAbstractTableModel.h>
#include <Wt/WString.h>

class PagedModel final : public Wt::WAbstractTableModel {
public:
  void setPage(PageResult<UserRow> p) {
    beginReset();
    page_ = std::move(p);
    endReset();
  }

  int rowCount(const Wt::WModelIndex& = Wt::WModelIndex()) const override {
    return static_cast<int>(page_.rows.size());
  }

  int columnCount(const Wt::WModelIndex& = Wt::WModelIndex()) const override {
    return 4;
  }

  Wt::cpp17::any data(const Wt::WModelIndex& index,
                      Wt::ItemDataRole role = Wt::ItemDataRole::Display) const override {
    if (!index.isValid() || role != Wt::ItemDataRole::Display) return {};
    const auto& r = page_.rows.at(static_cast<std::size_t>(index.row()));
    if (index.column() == 0) return Wt::WString::fromUTF8(std::to_string(r.id));
    if (index.column() == 1) return Wt::WString::fromUTF8(r.email);
    if (index.column() == 2) return Wt::WString::fromUTF8(r.displayName);
```

```
    return Wt::WString::fromUTF8(r.status);
  }

  const PageResult<UserRow>& page() const noexcept { return page_; }

private:
  PageResult<UserRow> page_{};
};
```

18.2.9 Pagination controls and UX behavior

A professional paging UX includes:

- disabled “Next” when there is no next page,
- disabled “Prev” on the first page,
- visible page position when total is available,
- consistent preservation of filters and sorting across page changes,
- loading indicators during refresh.

```
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

void updatePagerUi(const PageResult<UserRow>& p) {
  prevBtn_->setDisabled(!p.hasPrev);
  nextBtn_->setDisabled(!p.hasNext);

  if (p.totalRows >= 0) {
    const auto from = p.pageIndex * p.pageSize + 1;
    const auto to   = p.pageIndex * p.pageSize + static_cast<int>(p.rows.size());
    info_->setText(Wt::WString::fromUTF8(
```

```
      std::to_string(from) + "-" + std::to_string(to) +
      " of " + std::to_string(p.totalRows)));
  } else {
    info_->setText("");
  }
}
```

18.2.10 Sorting and filtering interaction with pagination

Sorting and filtering must reset paging state:

- when sort changes, reset to page 0,
- when filters/search change, reset to page 0,
- preserve user-selected page size unless policy forbids it.

```
void onSearchChanged() {
  query_.pageIndex = 0;
  reload();
}

void onSortChanged(SortSpec s) {
  query_.sort = std::move(s);
  query_.pageIndex = 0;
  reload();
}
```

18.2.11 Concurrency: what users see when data changes

Data can change between page requests. Professional handling:

- ensure stable ordering to reduce duplicates/missing rows,

- accept that the dataset may shift and design UI messaging for it,
- for critical workflows, provide snapshot semantics (rare) or explicit refresh cues.

For business-critical tables, you can provide:

- a “Refresh” button,
- warning when the table may be stale,
- conflict handling for row actions (edit/delete).

18.2.12 Performance rules for production pagination

Keep queries indexed

Pagination queries must use indexes aligned with:

- filtering conditions,
- sorting columns,
- and the tie-breaker key.

Load only needed columns

Do not fetch large text blobs or rarely used columns for list views.

Avoid N+1 loading

A table list view should not run one query per row for related data. If related data is needed:

- join carefully,
- prefetch in batch,
- or display a simplified list and load details on demand.

Cache stable reference data

Statuses, roles, and lookup tables can be cached to reduce repeated work, but data correctness must be preserved.

18.2.13 Professional checklist for Wt pagination

- Enforce strict page size bounds.
- Always apply deterministic ordering with a unique tie-breaker.
- Reset paging on filter/sort change.
- Keep only the current page in memory.
- Provide clear navigation state (prev/next, page info when available).
- Treat total counts as optional; avoid expensive counts per page.
- Choose offset pagination for simplicity, keyset pagination for large datasets.
- Ensure filters and authorization apply to every page query.
- Design for concurrency: stable ordering, refresh cues, conflict-safe actions.
- Keep queries efficient: indexed, bounded, minimal columns, no N+1.

18.2.14 Outcome

A professionally designed pagination system in Wt enables data tables to remain fast, stable, and predictable as data scales. It protects server resources, preserves UX responsiveness under latency, and provides the foundation for sorting, filtering, and row actions without performance collapse or correctness drift.

18.3 Filtering

Filtering is the mechanism that turns a table from a passive list into an operational tool. In production systems, filtering is not only a UI convenience; it is a correctness and performance feature. A well-designed filtering system:

- reduces cognitive load by narrowing results to what the user needs,
- enforces access control by ensuring users see only authorized data,
- protects performance by limiting result sets and query cost,
- and supports repeatable workflows by preserving filter state across navigation.

In Wt, filtering must be designed for server-side execution. The UI collects filter inputs, but the **service layer and repository layer are authoritative** for filter interpretation, normalization, validation, and query construction.

18.3.1 Filtering requirements in production-grade tables

A filtering system is professional when it provides:

- **clear semantics**: users understand what each filter means.
- **safe defaults**: empty filter means "show allowed rows", not "show everything unbounded".
- **composability**: multiple filters can be combined predictably.
- **stability**: filters persist during pagination and sorting.
- **validation**: invalid filter values are rejected safely and explained.
- **performance**: queries remain indexed and bounded.

- **security**: authorization constraints are always applied regardless of UI.

18.3.2 Filtering is a contract: UI → controller → service → repository

A reliable architecture follows this chain:

1. UI collects filter inputs (text boxes, dropdowns, date ranges).
2. Controller normalizes and validates the request shape (types, bounds).
3. Service applies business rules and authorization constraints.
4. Repository performs safe query construction and returns a page result.

UI widgets must never build SQL or access persistence directly.

18.3.3 Filter types you must support

Professional filtering typically includes the following categories.

Global text search

A single text box that searches multiple columns (e.g., email, name, id). It must be:

- bounded in length,
- normalized (trim, collapse whitespace),
- implemented safely to avoid injection,
- supported by indexes where possible.

Field-specific filters

Examples:

- status equals (Active/Locked),
- role equals (Admin/User),
- numeric ranges,
- exact-match ID lookup.

Field-specific filters provide strong predictability and performance.

Date/time range filters

These require careful semantics:

- define inclusive/exclusive boundaries consistently,
- ensure time zone handling is explicit,
- validate that end $\geq$ start,
- clamp ranges to prevent abusive wide scans.

Multi-select filters

Examples:

- status in {Active, Pending},
- category in {A, C}.

Multi-select filters must be bounded (max items) to keep query complexity controlled.

18.3.4 Designing a filter request object (DTO)

Use an explicit, typed request object for filtering. Avoid passing raw query strings across layers.

```
#include <string>
#include <vector>
#include <optional>
#include <chrono>

enum class SortDir { Asc, Desc };

struct SortSpec {
  std::string column;
  SortDir dir{SortDir::Asc};
};

struct DateRange {
  std::chrono::system_clock::time_point from;
  std::chrono::system_clock::time_point to;
};

struct UserFilter {
  std::string search;                  // global search
  std::optional<std::string> status;   // exact match
  std::optional<std::string> role;     // exact match
  std::optional<DateRange> created;    // date range
};

struct TableQuery {
  int pageIndex{0};
  int pageSize{50};
  std::optional<SortSpec> sort;
  UserFilter filter;
};
```

18.3.5 Normalization and validation (controller/service level)

Filtering must never trust UI input. Normalize and validate consistently:

- trim strings and collapse whitespace,
- enforce maximum lengths,
- validate enums (status/role),
- clamp page size and date ranges,
- reject unknown sort columns.

```
static void trim_in_place(std::string& s) {
  auto is_space = [](unsigned char c){ return c==' '||c=='\t'||c=='\n'||c=='\r'; };
  while (!s.empty() && is_space(static_cast<unsigned char>(s.front()))) s.erase(s.begin());
  while (!s.empty() && is_space(static_cast<unsigned char>(s.back())))  s.pop_back();
}

static std::string normalize_search(std::string s) {
  trim_in_place(s);
  if (s.size() > 128) s.resize(128); // hard safety bound
  return s;
}

static bool is_allowed_status(const std::string& s) {
  return s == "Active" || s == "Locked" || s == "Pending";
}

static bool is_allowed_sort_column(const std::string& c) {
  return c == "id" || c == "email" || c == "name" || c == "status" || c == "created";
}
```

Professional rule If a filter value is invalid:

- return a stable `InvalidInput` or validation error category,
- show a user-friendly message,
- do not attempt to guess what the user meant.

18.3.6 Filtering and authorization: never separate them

A users visible dataset must always be restricted by authorization constraints. These constraints are not "filters" chosen by the user, but they must be applied to every query:

- tenant isolation (multi-tenant apps),
- role restrictions,
- row-level security constraints.

Even if the UI sends no filters, the service must still restrict results to authorized rows.

18.3.7 Repository query construction: safety and performance

Safety: parameterization and whitelisting

Professional filtering requires:

- parameterized queries for values,
- whitelist mapping for columns (sorting and field selection),
- no raw concatenation of user input into SQL.

Performance: index-aligned filtering

Filters must be designed around indexes:

- equality filters are typically index-friendly,
- prefix searches can be index-friendly depending on DB and collation,
- leading-wildcard LIKE searches are typically expensive,
- date ranges should use indexed timestamps.

18.3.8 Wt UI patterns for professional filtering

1) Filtering controls

Common filter controls:

- search input (global),
- status dropdown,
- role dropdown,
- date range picker or two date inputs,
- reset button.

2) Preserve filter state across reloads

A professional table preserves filter state:

- when paginating,
- when sorting,

- when navigating away and coming back.

Store state in:

- internal path (bookmarkable),
- or a session-level view state object.

3) Debounce search requests

Global search should not trigger a server query on every keystroke. Use a delayed trigger approach:

- trigger on “Enter”,
- or trigger after a pause in typing,
- or trigger on blur.

```
search_->enterPressed().connect([this] {
  query_.pageIndex = 0;
  query_.filter.search = normalize_search(search_->text().toUTF8());
  reload();
});
```

4) Clear and reset behavior

Provide a reset mechanism that returns to a known safe baseline:

```
resetBtn_->clicked().connect([this] {
  query_.pageIndex = 0;
  query_.filter = UserFilter{};
  search_->setText("");
  statusCombo_->setCurrentIndex(0);
  roleCombo_->setCurrentIndex(0);
  reload();
});
```

18.3.9 Filtering interaction with pagination and sorting

Reset paging on filter change

Any filter change should reset to the first page:

```
void onFilterChanged() {
  query_.pageIndex = 0;
  reload();
}
```

Stable sorting is required

Filtering must always operate under a stable sort order. If a filter reduces results, the ordering must remain deterministic.

18.3.10 Empty states, partial matches, and user trust

Filtering UX must clearly communicate:

- when there are no results,
- what filters are active,
- how to broaden the search (reset or remove filters).

Example empty-state messaging:

- “No users match the current filters.”
- “Try clearing the status filter or searching by email.”

18.3.11 Testing filtering professionally

Filtering must be tested at multiple layers:

- service tests for normalization/validation rules,
- repository integration tests for query correctness and indexing behavior,
- UI tests for wiring (filter controls trigger reload and preserve state).

```
#include <cassert>

static void test_search_normalization() {
  auto s = normalize_search("   hello   ");
  assert(s == "hello");
}
```

18.3.12 Checklist for production-grade filtering

- Filters are represented as typed DTOs, not raw query strings.
- Inputs are normalized and validated (length bounds, enums, ranges).
- Sorting columns are whitelisted; values are parameterized.
- Authorization constraints are applied to every query.
- Filtering resets pagination and preserves stable ordering.
- Search is debounced; reset is available and predictable.
- Empty states are clear and help users recover.
- Filtering behavior is covered by service and repository tests.

18.3.13 Outcome

When filtering is engineered professionally in Wt, tables remain fast and predictable as data grows. Users can reliably narrow large datasets, the server remains protected from unbounded queries, and the architecture stays maintainable because filtering semantics live in services and repositories rather than being improvised inside UI callbacks.

18.4 Sorting

Sorting is the control that defines how users interpret and navigate data. In production systems, sorting is not a decorative feature: it is a correctness constraint, a performance concern, and a security boundary. In Wt, tables are server-driven, which means sorting must be designed as a cooperative pipeline between the UI, controller, service, and repository layers.
This section presents professional sorting techniques for Wt table views, emphasizing stable ordering, server-side sorting, safe column whitelisting, performance-aware design, and predictable UX behavior under pagination, filtering, and concurrent data changes.

18.4.1 What sorting must guarantee

A production-grade sorting system must guarantee:

- **determinism**: the same request produces the same ordering.
- **stability**: sorting has a stable tie-breaker to avoid duplicates/missing rows during pagination.
- **safety**: user-provided sort inputs cannot be used to inject SQL or access unauthorized columns.
- **performance**: sorting uses indexes and avoids sorting huge unbounded datasets in memory.

- **coherence**: sorting interacts predictably with filtering and paging.

18.4.2 Sorting as an explicit contract

Sorting should be represented as a typed specification, not as raw strings.

```
#include <string>

enum class SortDir { Asc, Desc };

struct SortSpec {
  std::string column; // must be validated and mapped to known fields
  SortDir dir{SortDir::Asc};
};
```

A table request typically includes paging and filtering too:

```
#include <optional>
#include <string>

struct TableQuery {
  int pageIndex{0};
  int pageSize{50};
  std::optional<SortSpec> sort;
  std::string search;
};
```

18.4.3 Server-side sorting is the default for production

Client-side sorting is only acceptable when:

- datasets are small,
- all data is loaded already,

- and security constraints are not violated by exposing full datasets.

In real systems, tables typically show a page of a much larger dataset. Therefore:

> **Sorting must be executed in the same place where paging and filtering are executed: the service/repository layer.**

18.4.4 Stable ordering: always add a tie-breaker

Sorting by non-unique columns (name, status, created date) produces ambiguous ordering. When combined with pagination, ambiguous ordering causes:

- duplicates across pages,
- missing rows,
- unpredictable navigation when rows are inserted/updated.

Professional sorting always produces a deterministic order by adding a unique tie-breaker:

- primary sort: requested column,
- secondary sort: `id` (or another unique key) in the same direction or a fixed direction.

```
struct NormalizedSort {
  std::string primary; // validated
  SortDir dir{SortDir::Asc};
  bool tieBreakWithId{true};
};
```

18.4.5 Sorting and security: whitelist, do not trust UI

Never accept arbitrary column names from the UI and pass them to SQL. The UI can only request a sort *key* that maps to a known set of sortable columns.

```
static bool is_allowed_sort_key(const std::string& k) {
  return k == "id" || k == "email" || k == "name" || k == "status" || k == "created";
}
```

Then map the key to an internal field:

```
static std::string map_sort_key_to_db_column(const std::string& k) {
  if (k == "id")      return "id";
  if (k == "email")   return "email";
  if (k == "name")    return "display_name";
  if (k == "status")  return "status";
  if (k == "created") return "created_at";
  return "id"; // safe fallback
}
```

18.4.6 Normalization rules for sorting

Professional systems normalize sorting requests so behavior is predictable:

- if no sort is requested, apply a default (often `created_at desc` or `id desc`),
- if an invalid sort key is provided, reject or fall back safely,
- always include a tie-breaker,
- reset pagination when sort changes.

```
static NormalizedSort normalize_sort(const std::optional<SortSpec>& sort) {
  NormalizedSort ns;
  if (!sort || !is_allowed_sort_key(sort->column)) {
    ns.primary = "id";
    ns.dir = SortDir::Desc;
    ns.tieBreakWithId = true;
    return ns;
```

```
  }
  ns.primary = sort->column;
  ns.dir = sort->dir;
  ns.tieBreakWithId = true;
  return ns;
}
```

18.4.7 Sorting UX: predictable interactions

A professional table sorting UX follows clear, consistent rules:

- clicking a column header toggles direction (Asc/Desc),
- only one primary sort is active unless multi-sort is explicitly supported,
- active sort state is visible (indicator arrow),
- sorting resets to page 0 because page boundaries shift under new ordering.

```
void onSortChanged(const std::string& key) {
  if (!query_.sort || query_.sort->column != key) {
    query_.sort = SortSpec{key, SortDir::Asc};
  } else {
    query_.sort->dir = (query_.sort->dir == SortDir::Asc) ? SortDir::Desc : SortDir::Asc;
  }
  query_.pageIndex = 0;
  reload();
}
```

18.4.8 Sorting and performance engineering

Index-aligned sorting

Sorting performance depends on indexes. A production system:

- sorts using indexed columns whenever possible,
- avoids sorting huge filtered datasets without indexes,
- keeps sorting and filtering aligned (composite indexes if needed).

Avoid sorting in memory

Sorting must not be done by loading all rows and sorting in C++ for large tables. This:

- increases memory usage,
- increases latency,
- breaks pagination,
- and risks exposing unauthorized data.

Limit selected columns

For list views:

- fetch only the columns displayed,
- do not fetch large text blobs,
- do not fetch heavy aggregates on the list query.

18.4.9 Sorting in combination with filtering and pagination

Sorting must interact correctly with filtering and pagination:

- filtering changes the dataset; keep the current sort unless the UI explicitly resets it,
- sorting changes the dataset order; reset paging to page 0,
- always ensure stable ordering across pages (tie-breaker).

18.4.10 Concurrency effects and user trust

When rows change during browsing:

- stable ordering reduces duplicates/missing rows but cannot eliminate all shifts,
- inform users when strong consistency is required (rare),
- provide refresh options for long-lived table sessions.

Conflict-safe row actions (edit/delete) should be protected by service-level concurrency checks (e.g., version fields) and mapped to user-friendly messages.

18.4.11 Multi-column sorting (advanced)

Multi-column sorting can be valuable but must be designed explicitly:

- define the maximum number of sort keys (e.g., 2–3),
- define UI interaction (shift-click to add secondary sort),
- keep stable tie-breaker as the final key.

```
struct MultiSortSpec {
  std::vector<SortSpec> keys; // validate length and allowed columns
};
```

If multi-sort is not required, do not implement it. Single-sort with a tie-breaker is simpler, safer, and faster.

18.4.12 Testing sorting professionally

Sorting logic must be tested at multiple layers:

- unit tests for sort normalization and allowed columns,
- integration tests for query correctness and index usage assumptions,
- UI tests to confirm sort indicators and page reset behavior.

```
#include <cassert>

static void test_sort_normalization_defaults() {
  auto ns = normalize_sort(std::nullopt);
  assert(ns.primary == "id");
  assert(ns.tieBreakWithId);
}
```

18.4.13 Checklist for production-grade sorting

- Sorting is represented as a typed contract (not raw strings).
- Sort keys are whitelisted and mapped to known internal columns.
- Services normalize sort requests and apply a safe default.
- Sorting is executed server-side with filtering and paging.
- Deterministic ordering is enforced using a unique tie-breaker.
- Sorting resets pagination to maintain coherent navigation.
- Sorting is index-aware and avoids in-memory sorting of large datasets.
- Sorting behavior is covered by unit and integration tests.

18.4.14 Outcome

Professional sorting design in Wt delivers predictable navigation, stable pagination behavior, and scalable performance as datasets grow. It keeps sorting safe from injection and unauthorized access, and it maintains maintainability by locating sorting semantics in services and repositories rather than distributing them across UI callbacks.

File Uploads and Downloads

19.1 File upload handling

File uploads are a common requirement in real-world web applications: user avatars, documents, images, logs, and bulk data imports. In Wt, file upload handling must be designed with explicit attention to security, performance, memory management, resilience, and user experience. Unlike simple form fields, file content cannot be trusted and must be processed defensively at every stage.
This section presents professional techniques for handling file uploads in Wt, including secure reception, size and type validation, streaming versus buffering, storage strategies, integration with services, error messaging, and safe persistence.

19.1.1 Server-side file handling principles

- **Never trust client input**: file name, content type, extensions, and size must be validated.
- **Avoid unbounded memory use**: large files must be streamed to disk or cloud storage rather than buffered in memory.
- **Security policy enforcement**: scan for malware if required, restrict upload locations, apply authorization.

- **Service-layer integration**: file metadata and business rules belong in services; UI merely collects input.
- **Clear user feedback**: upload progress and errors must be communicated reliably.
- **Consistent error mapping**: map storage and validation failures to stable UI outcomes.

19.1.2 Wt file upload controls

Wt provides `WFileUpload` widgets that trigger server-side upload events. Use them as a controlled input mechanism:

```
#include <Wt/WFileUpload.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

auto upload = addNew<Wt::WFileUpload>();
upload->setFileTextSize(40 * 1024); // show bytes
upload->setMaxFileSize(10 * 1024 * 1024); // hard limit: 10 MiB

auto status = addNew<Wt::WText>();
auto submitBtn = addNew<Wt::WPushButton>("Upload");

submitBtn->clicked().connect([this] {
  if (!upload->canUpload()) {
    status->setText("No file selected or file too large.");
    return;
  }
  upload->upload();
});
```

19.1.3 Size limits and resource bounds

1) Client-side size hints Setting `setMaxFileSize()` on the widget provides immediate feedback and prevents posts larger than the specified limit from even being sent by Wts built-in handler.

2) Server-side authoritative size enforcement The server must re-validate file size because client constraints can be bypassed.

```
if (upload->uploadedFiles().empty()) {
  throw std::runtime_error("Upload failed: no file received.");
}
auto fileInfo = upload->uploadedFiles().front();
if (fileInfo.spoolFileSize > static_cast<std::size_t>(10 * 1024 * 1024)) {
  throw std::runtime_error("Exceeded maximum server size.");
}
```

19.1.4 Allowed types and extension validation

Never trust content types and extensions from the client. Use server-side checks:

```
static bool is_allowed_extension(const std::string& name) {
  const auto dotPos = name.find_last_of('.');
  if (dotPos == std::string::npos) return false;
  const auto ext = name.substr(dotPos + 1);
  return ext == "jpg" || ext == "png" || ext == "pdf" || ext == "txt";
}
```

To improve safety, examine the files actual content (magic numbers) where feasible.

19.1.5 Streaming uploads

For large files, avoid full buffering in memory. Wts upload can be configured to spool to disk directly. This protects memory and enables processing in chunks:

```
upload->setProgressive(true);
```

Streaming mode allows the server to receive the file in pieces and write it incrementally to a temporary location.

19.1.6 Integrating upload with services

Treat file uploads as a two-step integration:

- the controller/handler receives and validates the raw file,
- the service processes the file content, stores metadata, and enforces business rules.

```
struct UploadedFileMetadata {
  std::string storedPath;
  std::string originalName;
  std::size_t size{};
};

struct IFileService {
  virtual ~IFileService() = default;
  virtual Result<UploadedFileMetadata> store_user_file(
    std::int64_t userId,
    const std::string& tempPath,
    const std::string& originalName,
    std::size_t size) = 0;
};
```

This abstraction keeps UI and controller free from storage concerns and business policy.

19.1.7 Safe storage strategies

Local filesystem

Store files outside the web root. Do not trust user-provided names:

```
#include <filesystem>
#include <chrono>

std::string make_unique_filename(const std::string& original) {
  auto ts = std::chrono::system_clock::now().time_since_epoch().count();
  return std::to_string(ts) + "-" + original;
}

std::string store_temp_file(const std::string& tempPath,
                            const std::string& userName) {
  const auto safeName = make_unique_filename(userName);
  std::filesystem::path target = "/var/app/uploads/" / safeName;
  std::filesystem::rename(tempPath, target);
  return target.string();
}
```

Avoid collisions and use a well-defined root directory with strict permissions.

Cloud/object storage

For high-scale systems, upload files directly to object storage (S3/compatible) via pre-signed URLs or server-side transfer after validation. Ensure metadata and authorization are consistent.

19.1.8 Virus/malware scanning

If uploads can carry untrusted content, integrate scanning tools. The server should:

- scan the file before persisting in user-accessible paths,

- quarantine or reject infected files,
- log scan results for audit.

Scanning can be synchronous for small files or asynchronous for large files, with status updates provided to users.

19.1.9 Progress feedback and UX

Uploads can be slow; users must see progress. Wts `WFileUpload` provides events:

```
upload->fileTooLarge().connect([this] {
  status->setText("File exceeds the allowed maximum size.");
});

upload->uploaded().connect([this] {
  status->setText("Upload complete. Processing...");
});

upload->progress().connect([this](std::size_t done, std::size_t total) {
  status->setText("Progress: " + std::to_string(done) + "/" + std::to_string(total));
});
```

These events improve perceived responsiveness and reduce user frustration.

19.1.10 Error handling during upload

Handle errors categorically:

- client-side violations (size/type),
- server-side validation failures,
- storage write errors,

- service-level business policy violations.

Map each to clear user-facing messages and stable error categories.

```
try {
  upload->upload();
} catch (const std::exception& e) {
  status->setText("Upload failed: " + Wt::WString::fromUTF8(e.what()));
}
```

Server-side handlers should catch exceptions and translate them into structured Fail results when invoking services.

19.1.11 Processing uploaded content

Once stored, process the file in services as needed:

- extract metadata,
- generate thumbnails or previews,
- parse content for import,
- associate with domain entities.

Processing can be synchronous (small files) or queued as a background job (large files or expensive operations).

19.1.12 Security boundary: access control for uploaded files

Files stored on behalf of a user must not be universally accessible. Apply:

- per-user or role-based access checks,

- signed URLs or temporary tokens for direct download,
- secret directories outside the public web root.

Serve files through handlers that enforce authorization:

```
void serve_user_file(const Wt::Http::Request& req, Wt::Http::Response& resp) {
  auto userId = get_session_user_id();
  auto fileId = parse_file_id(req);

  if (!authz_can_access(userId, fileId)) {
    resp.setStatus(403);
    return;
  }
  resp.out() << Wt::WFileUtil::readFile(get_storage_path(fileId));
}
```

Never serve user files directly from static locations without checks.

19.1.13 Testing file upload handling

Test file upload handling at multiple layers:

- controller-level tests with fake upload data,
- service tests verifying storage rules and business policy,
- integration tests for actual file system operations,
- error path tests (size violation, type violation).

```
#include <cassert>

static void test_extension_validation() {
  assert(is_allowed_extension("image.jpg"));
  assert(!is_allowed_extension("image.exe"));
}
```

19.1.14 Checklist for production-grade file uploads

- Enforce hard server-side size limits.
- Validate type/extension and content.
- Avoid buffering large files in memory; stream to disk/object storage.
- Integrate with services for business rules and persistence.
- Provide upload progress and clear feedback.
- Secure storage outside public web roots.
- Apply access control on download/serve paths.
- Handle all error categories explicitly.
- Scan for malware where appropriate.
- Test upload, validation, storage, and error paths thoroughly.

19.1.15 Outcome

Professional file upload handling in a Wt application ensures that users can send files safely, that system resources are protected, and that business rules and security policies are enforced consistently. By separating UI collection, controller normalization, service policy enforcement, and storage mechanics, the design remains testable, maintainable, and scalable as the application grows.

19.2 Security considerations

File uploads and downloads are among the highest-risk features in any web application. They combine untrusted input, potentially large payloads, storage concerns, and user-visible distribution channels. In a server-side Wt application, the same risk profile exists as in any web stack: attackers can upload crafted files, attempt path traversal, exploit parsers, trigger resource exhaustion, smuggle malicious payloads, or use downloads to bypass authorization boundaries. This section describes security-grade engineering practices for handling uploads and downloads safely in production Wt systems. The principles apply regardless of storage backend (local filesystem, network storage, object storage) and regardless of file type (images, PDFs, documents, archives, data imports).

19.2.1 Threat model for file features

A professional design begins by explicitly acknowledging the threat model:

- **Untrusted metadata**: file name, MIME type, extension, and client claims are not reliable.
- **Untrusted content**: file bytes can contain malware, parser exploits, polyglots, or decompression bombs.
- **Resource exhaustion**: oversized uploads, many uploads, slow uploads, and repeated retries can exhaust CPU, memory, disk, and database connections.
- **Privilege abuse**: users may attempt to access files belonging to other users or tenants.
- **Content distribution**: a download endpoint can become a malware distribution channel if not controlled.
- **Injection channels**: file names and metadata can be used for XSS, header injection, log poisoning, or command injection if mishandled.

19.2.2 Core security principles

Principle 1: Treat all uploaded files as hostile

Never assume any file is safe based on:

- extension,
- MIME type,
- source,
- or size.

Principle 2: Enforce strict bounds

Always enforce:

- maximum file size per upload,
- maximum total upload size per request,
- maximum number of files,
- rate limits per user and per IP,
- timeouts for slow uploads.

Principle 3: Store outside public paths

Never store user uploads in a directory served as static web content. Files should be stored:

- outside the web root,
- with restrictive filesystem permissions,
- using safe generated names.

Principle 4: Serve downloads through authorization checks

Files must be served through endpoints that enforce:

- user identity,
- tenant boundaries,
- and per-file authorization.

19.2.3 Upload security controls

1) Hard size limits (client and server)

Client-side hints improve UX, but server-side enforcement is mandatory.

```
constexpr std::size_t kMaxUploadSize = 10 * 1024 * 1024; // 10 MiB

upload_->setMaxFileSize(kMaxUploadSize);

upload_->uploaded().connect([this] {
  if (upload_->uploadedFiles().empty()) {
    status_->setText("No file received.");
    return;
  }
  const auto f = upload_->uploadedFiles().front();
  if (f.spoolFileSize > kMaxUploadSize) {
    status_->setText("File exceeds allowed size.");
    return;
  }
});
```

Also enforce request-level limits and rate limits in your HTTP layer (reverse proxy and/or application server policy).

2) File count limits and concurrency limits

Prevent abuse by limiting:

- number of concurrent uploads per session,
- number of files per request,
- total files stored per user (quota).

3) Extension and content verification

Never trust the extension alone. Use multiple checks:

- extension allow-list,
- MIME-type allow-list (advisory only),
- signature (magic number) checks where feasible,
- deep parsing only in controlled environments.

```
static bool has_allowed_extension(const std::string& name) {
  const auto dot = name.find_last_of('.');
  if (dot == std::string::npos) return false;
  const auto ext = name.substr(dot + 1);
  return ext == "png" || ext == "jpg" || ext == "pdf" || ext == "txt";
}
```

When you accept complex formats (office docs, archives), assume parser risk and isolate processing.

4) Safe filename handling

Client-provided file names can include:

- path separators,
- Unicode tricks,
- control characters,
- sequences that cause header injection if reused.

Professional rule:

> **Never use the original filename as a storage path. Use generated names, store the original name only as metadata.**

```
#include <string>
#include <chrono>

static std::string make_storage_name(std::string ext) {
  auto ts = std::chrono::system_clock::now().time_since_epoch().count();
  if (!ext.empty() && ext[0] != '.') ext = "." + ext;
  return std::to_string(ts) + ext;
}
```

5) Path traversal prevention

All file paths must be constructed from trusted roots and safe generated components.

```
#include <filesystem>

static std::filesystem::path safe_join(const std::filesystem::path& root,
                                       const std::filesystem::path& name) {
  auto p = root / name;
```

```
  p = p.lexically_normal();
  // Ensure the final path stays inside root:
  if (p.native().find(root.lexically_normal().native()) != 0) {
    throw std::runtime_error("Path traversal attempt.");
  }
  return p;
}
```

6) Malware scanning and quarantine workflow

If users can upload content that may be redistributed or opened by staff:

- scan files before making them accessible,
- quarantine suspicious files,
- store scan result and timestamp for audit,
- fail closed: do not serve unscanned files if policy requires scanning.

For large files, scanning may be asynchronous:

- upload accepted,
- file stored in quarantine,
- background job scans,
- file promoted to accessible state only after passing.

7) Content processing sandboxing

Processing uploads (image resizing, PDF parsing, archive extraction, data import) is a common exploitation path. Professional safety measures include:

- run parsers in isolated processes,
- apply strict timeouts and memory limits,
- avoid running external commands with untrusted parameters,
- restrict supported formats.

8) Zip bombs and decompression limits

If you accept archives:

- limit the number of entries,
- limit total expanded size,
- limit nesting depth,
- reject archives with suspicious compression ratios.

19.2.4 Download security controls

1) Authorization checks on every request

Never assume that knowledge of a file ID is sufficient. Every download must verify:

- authentication (who is requesting),
- authorization (are they allowed),
- tenant ownership (for multi-tenant systems).

```
void handleDownload(const Wt::Http::Request& req, Wt::Http::Response& resp) {
  const auto userId = session_user_id();
  const auto fileId = parse_file_id(req);
```

```
  if (!authz_can_access_file(userId, fileId)) {
    resp.setStatus(403);
    return;
  }

  // Serve file through controlled streaming.
  stream_file_to_response(file_storage_path(fileId), resp);
}
```

2) Do not leak file existence

Error responses should avoid confirming whether a file exists when unauthorized. A common policy:

- return 404 for unauthorized access to reduce enumeration,
- or return 403 consistently depending on your security policy.

3) Safe headers and content disposition

When serving user uploads, you typically want:

- `Content-Disposition: attachment` for untrusted types to avoid inline execution,
- safe filename encoding to avoid header injection.

```
resp.addHeader("Content-Disposition", "attachment; filename=\"download.bin\"");
resp.addHeader("X-Content-Type-Options", "nosniff");
```

Do not reflect the original filename into headers without sanitization.

4) Streaming to avoid memory exhaustion

Downloads should be streamed, not buffered into memory, especially for large files.

5) Signed URLs and expiring tokens (advanced)

For object storage backends, an industry-standard approach is:

- authorize in the application,
- issue a short-lived signed URL,
- let the client download directly from storage.

This reduces load on the application server while preserving authorization semantics.

19.2.5 Cross-cutting concerns

Audit logging

For compliance and incident response, log:

- who uploaded what and when,
- who downloaded what and when,
- file size, hash (optional), scan result,
- relevant request metadata (without storing sensitive content in logs).

Rate limiting and abuse prevention

File endpoints are attractive for abuse. Apply:

- per-IP and per-user upload rate limits,
- concurrency limits,
- storage quotas,
- anomaly detection (sudden spikes).

Integrity checks

Store integrity metadata:

- cryptographic hashes for important documents,
- content-length checks,
- and versioning where needed.

19.2.6 Common mistakes to avoid

- Accepting “MIME type” as proof of file type.
- Using original filenames as storage paths.
- Storing uploads in a publicly served static directory.
- Serving downloads without authorization checks.
- Parsing complex formats in-process without limits.
- Allowing unbounded page sizes or request sizes.
- Reflecting user-provided names into headers.
- Forgetting to secure temporary upload directories.

19.2.7 Testing security for file features

Security cannot rely on manual inspection. Tests should cover:

- size limit enforcement,
- extension/type allow-list enforcement,

- path traversal attempts,
- unauthorized download attempts,
- concurrency and rate-limit behavior,
- failure handling (storage errors, scan failures).

```
#include <cassert>

static void test_extension_policy() {
  assert(has_allowed_extension("a.pdf"));
  assert(!has_allowed_extension("../a.pdf"));
  assert(!has_allowed_extension("evil.exe"));
}
```

19.2.8 Checklist: secure uploads and downloads

- Enforce strict size, count, and rate limits.
- Validate extension and content; do not trust client metadata.
- Generate safe storage names; store outside public paths.
- Prevent path traversal by construction and verification.
- Scan and quarantine untrusted files where required.
- Isolate risky parsing in sandboxed processes with limits.
- Enforce authorization on every download request.
- Use safe headers and prefer attachment disposition for untrusted types.
- Stream uploads/downloads to avoid memory exhaustion.
- Audit log file operations and test all failure paths.

19.2.9 Outcome

When security considerations are engineered correctly, file upload and download features become reliable and defensible rather than a permanent vulnerability surface. The application gains strong resistance against injection, traversal, malware distribution, resource exhaustion, and unauthorized access, while still providing a usable and scalable experience for legitimate users.

19.3 Storage strategies

File storage is not an implementation detail; it is a core architectural decision that influences security, scalability, cost, backup strategy, compliance, and user experience. In production systems, the storage strategy must be selected based on:

- file sizes and throughput,
- durability and availability requirements,
- access patterns (frequent reads vs rare downloads),
- security and multi-tenant isolation,
- operational constraints (backup, restore, disaster recovery),
- and regulatory requirements (retention, deletion guarantees, auditability).

In Wt applications, the UI layer should never know *where* files are stored. It should only interact with a service interface that accepts uploaded data (or references to temporary upload spools) and returns a stable file identifier. The storage backend remains an implementation detail behind the service boundary.

19.3.1 Core design goals for storage

A professional storage strategy must provide:

- **security by default**: store outside public paths, enforce authorization for every access.
- **durability**: files persist reliably across crashes and deployments.
- **scalability**: storage grows without forcing application redesign.
- **recoverability**: backup/restore procedures are tested and predictable.
- **integrity**: corruption is detectable (hashing/versioning where needed).
- **traceability**: uploads/downloads are auditable.

19.3.2 A uniform storage abstraction

The service layer should expose a storage abstraction such that the rest of the application does not depend on the backend.

```
#include <cstdint>
#include <string>
#include <vector>
#include <optional>

struct StoredFile {
  std::int64_t id{};
  std::string storageKey;     // backend key, not exposed to clients
  std::string originalName;   // metadata
  std::string contentType;    // metadata (advisory)
  std::uint64_t sizeBytes{};
  std::string sha256;         // optional integrity metadata
};
```

```
struct StoreRequest {
  std::string tempPath;       // path to server-side spool file
  std::string originalName;
  std::string contentType;
  std::uint64_t sizeBytes{};
};

struct IFileStorage {
  virtual ~IFileStorage() = default;

  virtual StoredFile store(const StoreRequest& req) = 0;
  virtual std::optional<std::string> resolve_path(std::int64_t fileId) = 0;
  virtual void erase(std::int64_t fileId) = 0;
};
```

The UI and controllers communicate only via file IDs and service results. Storage keys, paths, buckets, and credentials remain internal.

19.3.3 Storage option 1: Local filesystem storage

When it fits

Local filesystem storage is suitable when:

- deployment is on a single host (or a tightly managed cluster with shared storage),
- files are moderate in size and volume,
- backup and access controls are manageable,
- operational simplicity is a priority.

Key requirements

- Store outside the web root.
- Use generated storage names, never user-provided names.
- Use strict permissions and a dedicated OS user.
- Avoid placing uploads in ephemeral directories that get cleaned unexpectedly.
- Consider hashing and directory sharding to avoid too many files per directory.

Directory layout strategy

A professional filesystem layout avoids millions of files in one directory. Use sharding:

- shard by the first bytes of a hash,
- or shard by date (year/month/day) for append-heavy workloads,
- or shard by tenant/user ID ranges for isolation.

```
#include <filesystem>
#include <string>

static std::filesystem::path shard_path(const std::filesystem::path& root,
                                        const std::string& hexHash) {
  // hexHash assumed at least 4 chars
  return root / hexHash.substr(0,2) / hexHash.substr(2,2);
}
```

Atomic store pattern

Avoid partially written files being visible:

- write to a temporary name,
- fsync when needed (policy-driven),
- rename into final name atomically.

```
#include <filesystem>

static void atomic_move(const std::filesystem::path& tmp,
                        const std::filesystem::path& final) {
  std::filesystem::create_directories(final.parent_path());
  std::filesystem::rename(tmp, final); // atomic on same filesystem
}
```

Pros and cons

- **Pros**: simple, fast, cheap, no external dependency.
- **Cons**: scaling across multiple servers is hard; backup/restore must be handled carefully; storage can become a single point of failure if not replicated.

19.3.4 Storage option 2: Shared network storage (NFS/SMB or distributed FS)

When it fits

Shared storage fits when:

- multiple application instances must access the same files,
- you want a single storage namespace,
- you have operations support to manage reliability.

Engineering considerations

- Network storage introduces latency; use caching for frequently accessed files.
- Ensure atomicity semantics match your assumptions (rename guarantees vary).
- Handle transient IO failures with retry policies.
- Monitor and enforce quotas to prevent storage exhaustion.

Pros and cons

- **Pros**: easy multi-instance access; central backup.
- **Cons**: network latency; operational complexity; potential contention and outages.

19.3.5 Storage option 3: Database BLOB storage

When it fits

Storing files as BLOBs inside a database can fit when:

- files are small (avatars, small PDFs),
- transactional coupling is important,
- you want unified backup/restore (data + file in one snapshot),
- access patterns are not extremely heavy.

Professional cautions

- Large BLOBs can bloat backups and slow replication.
- Database IO is expensive compared to object storage for large files.

- Streaming reads/writes must be used to avoid memory spikes.
- Consider storing only the metadata in DB and the content in object storage for larger workloads.

Pros and cons

- **Pros**: strong transactional semantics; simpler consistency model.
- **Cons**: scaling and cost; performance issues for large volumes; operational risk if DB becomes both data and file store.

19.3.6 Storage option 4: Object storage (S3-compatible)

When it fits

Object storage is the most common modern choice when:

- file volume is large or unpredictable,
- you need high durability and global availability,
- you want to offload serving downloads from app servers,
- you need lifecycle policies (retention, tiering, deletion).

Integration patterns

Two professional patterns dominate:

A) Server-mediated upload

- upload arrives at Wt server,

- server validates and scans,
- server uploads to object storage,
- server stores metadata in DB.

B) Direct-to-storage upload (pre-signed)

- client requests an upload authorization token,
- server returns a short-lived signed upload URL,
- client uploads directly to object storage,
- server finalizes metadata and state once upload is confirmed.

Pattern (B) reduces load on the application server and scales well.

Serving downloads

Professional download serving often uses:

- application-level authorization,
- then a short-lived signed download URL from storage.

This preserves authorization while allowing storage/CDN to deliver bytes efficiently.

Pros and cons

- **Pros**: scalable, durable, cost-effective for large data; lifecycle management; CDN compatibility.
- **Cons**: additional dependency; eventual consistency considerations in some systems; requires careful key naming and access control policies.

19.3.7 Metadata storage and file identity

Regardless of backend, production systems typically store metadata in a relational database:

- file ID (internal),
- storage key/path,
- original name,
- size,
- content type (advisory),
- owner user/tenant,
- upload timestamp,
- scan status,
- hash (optional),
- retention/deletion markers.

Why metadata belongs in DB

- authorization checks are queryable,
- searching and auditing are easy,
- business workflows (approval, review, lifecycle) are manageable,
- deletion and retention policies are enforceable.

19.3.8 Integrity and immutability strategy

Many file systems treat stored user uploads as immutable objects:

- a new upload creates a new stored object,
- updates create a new version rather than modifying the old object,
- metadata links the new version to the domain entity.

This simplifies:

- caching,
- auditing,
- rollback,
- concurrency handling.

19.3.9 Retention, deletion, and compliance

Storage strategy must include lifecycle management:

- retention periods (keep for N days/months/years),
- legal holds (prevent deletion),
- user-initiated deletion (and what “deletion” means operationally),
- secure deletion policy (when required),
- soft-delete vs hard-delete workflows.

A common pattern:

- mark metadata as deleted,
- schedule background deletion from storage,
- maintain audit logs.

19.3.10 Backup and disaster recovery considerations

Filesystem/shared storage

- ensure backups include both metadata DB and file store,
- test restore procedures, not only backup creation,
- define acceptable RPO/RTO (data loss and recovery time).

Object storage

- enable versioning where required,
- use cross-region replication for high availability,
- back up metadata DB; object storage durability does not replace metadata recoverability.

19.3.11 A practical selection guide

Choose based on scale and operational reality:

- **Single-server, small volume**: local filesystem + DB metadata.
- **Multi-instance, moderate volume**: shared storage + DB metadata (with strong monitoring).
- **Small files, strong transactional coupling**: DB BLOB (carefully bounded).

- **Large volume, scalable production**: object storage + DB metadata (often with signed URLs and CDN).

19.3.12 Implementation checklist

- UI never depends on storage backend; all access goes through services.
- Always store files outside public web roots.
- Use generated storage keys; store original names only as metadata.
- Use directory sharding for filesystem storage.
- Stream uploads/downloads to avoid memory exhaustion.
- Keep metadata in DB for authorization, audit, search, lifecycle.
- Define retention/deletion workflow and test it.
- Define backup/restore procedures and test them regularly.
- Use integrity hashes for critical documents.
- Prefer immutable objects and versioning for reliability.

19.3.13 Outcome

A well-chosen storage strategy makes file uploads and downloads reliable, secure, and scalable. By separating metadata from content storage, enforcing strict access control, and planning for lifecycle and disaster recovery, Wt applications can support real-world file workloads without accumulating security risk or operational fragility.

Part IX

Performance, Security, and Stability

Performance in Wt Applications

20.1 Memory management

Memory management is a critical performance concern in server-side applications built with Wt. Unlike client-side JavaScript frameworks where the browser handles much of the memory lifecycle, a Wt application runs as a native C++ process where memory usage directly affects throughput, stability, latency, and scalability. Professional memory management must be deliberate, efficient, and observable. This section explains best practices, common pitfalls, and patterns for robust memory handling in Wt applications.

20.1.1 Understanding the Wt execution model

Wt applications are multi-session, event-driven servers:

- each HTTP request, WebSocket message, or user event can allocate memory,
- the Wt internal widget tree and sessions remain alive across multiple requests,
- long-lived objects like sessions, caches, and models persist in memory,
- pooled threads and asynchronous operations add complexity to memory lifetimes.

Memory management must consider:

- object lifetime (stack vs heap),
- ownership (who is responsible for deallocation),
- fragmentation and leaks,
- caching and retention,
- concurrency and thread-safety.

20.1.2 Rule 1: Prefer automatic storage and RAII

C++s Resource Acquisition Is Initialization (RAII) is the foundation of safe memory management. Whenever possible:

- use automatic storage (local variables),
- use standard containers (`std::vector`, `std::string`, `std::unique_ptr`),
- prefer `std::unique_ptr` or `std::shared_ptr` for heap objects with dynamic lifetime.

RAII ensures memory is freed when the owning scope is exited, reducing leaks and simplifying reasoning.

20.1.3 Rule 2: Minimize dynamic allocations in hot paths

Dynamic memory allocation (`new`, `malloc`) is expensive in terms of CPU and fragmentation. In performance-critical sections such as:

- request handlers,
- serialization/deserialization,
- table view models,

- streaming data,

minimize unnecessary allocations. Techniques include:

- reuse pre-allocated buffers,
- use pools or arenas for many small objects,
- avoid temporary strings where possible.

```
#include <vector>
static std::vector<char> buffer;
buffer.reserve(8 * 1024); // reuse an 8KiB buffer across calls
```

20.1.4 Rule 3: Manage session and widget lifetimes explicitly

Wt retains objects for each session. Widgets and session-specific data can accumulate over time if not pruned. Professional patterns:

- remove widgets when no longer needed (`removeChild()`),
- clear large containers explicitly (`clear()`),
- avoid storing unbounded history in sessions.

Long-lived per-session state should be bounded and reset as appropriate.

20.1.5 Rule 4: Use smart pointers for ownership clarity

Smart pointers express ownership intent:

- `std::unique_ptr` for exclusive ownership,
- `std::shared_ptr` for shared ownership when necessary,
- avoid `std::shared_ptr` cycles (use `std::weak_ptr` for back references).

Avoid raw owning pointers unless deeply justified by performance budgets.

20.1.6 Rule 5: Avoid global state that accumulates memory

Global or static containers can inadvertently accumulate memory over time. Instead:

- use scoped resources,
- encapsulate global caches behind eviction policies,
- monitor memory usage patterns over time.

Large global caches without eviction can cause OOM crashes under sustained load.

20.1.7 Rule 6: Apply object pooling for frequent allocations

Objects that are repeatedly allocated and freed (e.g., per-request structs) can benefit from pooling. Simple pool pattern:

```
#include <stack>
#include <memory>

template <typename T>
class ObjectPool {
public:
  std::unique_ptr<T> acquire() {
    if (!pool_.empty()) {
      auto obj = std::move(pool_.top());
      pool_.pop();
      return obj;
    }
    return std::make_unique<T>();
  }

  void release(std::unique_ptr<T> obj) {
    pool_.push(std::move(obj));
```

```
  }

private:
  std::stack<std::unique_ptr<T>> pool_;
};
```

Pools improve performance when objects are homogeneous and short-lived.

20.1.8 Rule 7: Use move semantics to avoid copies

C++11 move semantics eliminate unnecessary deep copies. Always prefer:

- returning by value with `std::move`,
- passing and storing `std::string` and containers by value and moving them,
- avoiding const reference when ownership must transfer.

```
std::string make_response() {
  std::string out = compute();
  return out; // optimized with RVO/move
}
```

20.1.9 Rule 8: Profile and measure memory usage

Assumptions about memory usage can be wrong under load. Use professional tools:

- heap profilers (Valgrind massif, heaptrack),
- OS-level memory monitors,
- application-level instrumentation (alloc counts, peak memory).

Measure before and after optimization. Blind optimization can reduce clarity without benefit.

20.1.10 Rule 9: Prevent leaks across asynchronous tasks

Wt applications often schedule asynchronous tasks:

- database queries off the main thread,
- background processing queues,
- deferred cleanups.

Use scopes or smart pointers to ensure task data is freed when the task completes. Avoid capturing large state in lambdas that outlive the request:

```
auto data = std::make_shared<LargeStruct>();
dispatch_async([data] {
  process(*data);
}); // 'data' is freed automatically when no longer used
```

20.1.11 Rule 10: Avoid accidental retention in closures

Closures capturing `this` or member pointers can unintentionally extend lifetimes. Prefer capturing only what is needed:

```
auto widgetPtr = widget.get();
lambda_capture([widgetPtr] { widgetPtr->doSomething(); });
```

This avoids prolonged ownership via `shared_ptr` cycles or accidental retention of session scope.

20.1.12 Rule 11: Bound caches with eviction policies

Caches improve performance but can grow unbounded. Implement:

- size or entry count limits,

- time-based expiry,
- least-recently-used (LRU) or other eviction strategies.

```
#include <unordered_map>
#include <list>

template <class K, class V>
class LRUCache {
  // Implementation omitted for brevity; core idea is to
  // keep recent keys in front and evict oldest when size limit hit.
};
```

Unbounded caches can exhaust memory under long runs.

20.1.13 Rule 12: Watch fragmentation and defragmentation

Native heap fragmentation can occur under mixed allocation sizes. Strategies:

- pool small objects,
- allocate large buffers once and reuse,
- tune allocator parameters (jemalloc, tcmalloc) for server workloads.

Use professional allocators if native fragmentation becomes problematic.

20.1.14 Rule 13: Streaming large payloads instead of buffering

Large file uploads, downloads, or data exports should never be fully buffered:

- stream in chunks,
- process incrementally,

- apply backpressure,
- avoid loading entire content into memory.

Wt supports streamed responses; use it to avoid OOM on large transfers.

20.1.15 Rule 14: Guard against retain cycles

Smart pointers can cause retain cycles when two objects share ownership. Break cycles with `std::weak_ptr`:

```
std::shared_ptr<A> a;
std::shared_ptr<B> b;
a->bptr = b;
b->aptr = a; // cycle
// Instead use:
b->aptr = a; // shared; and
a->bptr = std::weak_ptr<B>(b); // weak breaks cycle
```

Retain cycles are subtle and cause long-term leaks in persistent servers.

20.1.16 Rule 15: Monitor memory usage in production

Memory usage must be observable:

- exposure via health endpoints,
- logs with memory high-water marks,
- alerts on unusual growth,
- integration with monitoring stacks (Prometheus, etc.).

Proactive observation prevents slow leaks from becoming outages.

20.1.17 Testing memory behavior

Tests should include:

- unit tests for object lifetime,
- integration tests for cumulative memory under simulated load,
- stress tests with tools to detect leaks and fragmentation,
- validations on memory guards (bounds, eviction).

```
#include <cassert>
static void test_unique_ptr_cleanup() {
  std::unique_ptr<int> p = std::make_unique<int>(42);
  p.reset();
  assert(p == nullptr);
}
```

20.1.18 Checklist for memory-safe Wt applications

- Prefer RAII and smart pointers for ownership.
- Minimize dynamic allocations in hot paths.
- Bound session and cache memory.
- Use object pools where appropriate.
- Apply move semantics to avoid copies.
- Stream large payloads to avoid buffering.
- Avoid global unbounded state.

- Profile memory usage regularly.
- Break retain cycles with weak pointers.
- Monitor memory in production with alerts.

20.1.19 Outcome

When memory management is engineered professionally, Wt applications remain stable under sustained traffic, avoid slow leaks, maintain predictable latency, and scale with confidence. Correct ownership, bounded allocations, and observability ensure that memory does not become a silent cause of degradation or failure.

20.2 Session scalability

Session scalability is the defining performance challenge of server-side UI frameworks. In Wt, each connected user is represented by a server-side session that owns a `Wt::WApplication` instance and a widget tree. This delivers strong consistency and a powerful programming model, but it also means that scaling is not only about request throughput; it is about controlling *per-session memory*, *per-session CPU work*, *session lifetime*, and *cross-process routing*.
A production-grade Wt architecture must treat sessions as a managed resource:

- keep per-session state minimal and bounded,
- avoid long-lived expensive objects per session,
- ensure timeouts and cleanup are correct,
- design for horizontal scaling (multiple workers/nodes),
- and maintain predictable latency under concurrent sessions.

20.2.1 What “session scalability” really means

In a Wt system, session scalability means you can increase the number of concurrent users without:

- memory growth proportional to session count (or at least not exploding),
- blocking contention on shared resources (DB connections, locks, caches),
- UI lag caused by session thread starvation,
- server instability caused by long-lived sessions or retained widget trees,
- failure to scale horizontally due to session routing constraints.

A session-scalable Wt application can handle:

- many active sessions with moderate UI complexity,
- many idle sessions with minimal resource footprint,
- spikes in activity without degrading system-wide performance.

20.2.2 The session cost model

A session has a recurring cost in:

- **memory**: widget tree, models, cached data, uploaded spools, user context,
- **CPU**: event processing, model updates, rendering diffs, server-side validation,
- **shared resources**: DB connections, file handles, caches, thread pool capacity.

Therefore, session scalability is achieved by controlling:

- **per-session footprint** (memory and retained objects),

- **per-event work** (CPU per click / per update),
- **session lifetime and churn** (creation/destruction overhead),
- **resource multiplexing** (sharing resources safely across sessions).

20.2.3 Rule 1: Keep per-session state minimal and bounded

The most common scalability failure in server-side UI apps is storing too much per session:

- large query result sets,
- entire domain objects instead of IDs,
- unbounded logs/history in memory,
- cached binary content (images, PDFs),
- large widget subtrees that are never released.

Professional guidance:

- store *identifiers* and *small view state*, not full datasets,
- store only the current page for tables (paging), not the entire table,
- clear temporary buffers immediately after use,
- detach and delete widgets that are no longer visible or relevant.

```
struct SessionViewState {
  int currentPage{0};
  int pageSize{50};
  std::string search;
  // Store IDs, not full domain objects:
  std::vector<std::int64_t> selectedIds;
};
```

20.2.4 Rule 2: Prefer stateless services; avoid per-session heavy dependencies

Services that talk to databases, caches, or storage should generally be:

- shared across sessions (stateless), or
- lightweight objects with references to shared pools.

Anti-pattern:

- creating one DB connection per session,
- creating one HTTP client per session with its own thread resources,
- creating large caches per session.

Instead:

- share services and allocate from pools,
- keep per-session objects small (auth context, small view state).

```
class DbPool {
public:
  // Acquire/release pooled connections; details omitted.
};

class UserService {
public:
  explicit UserService(DbPool& pool) : pool_(pool) {}
private:
  DbPool& pool_;
};
```

20.2.5 Rule 3: Use session timeouts and cleanup hooks correctly

Sessions that never expire will eventually consume resources, even if each is small. Production requires:

- idle session timeouts,
- explicit cleanup of large session-owned resources,
- expiration behavior that is predictable to the user.

When a session ends, ensure you release:

- temporary files,
- cached objects,
- background tasks associated with the session,
- pending subscriptions.

```
class MyApp : public Wt::WApplication {
public:
  MyApp(const Wt::WEnvironment& env) : Wt::WApplication(env) {
    // ...
  }

  ~MyApp() override {
    // Release session-owned resources here:
    // - cancel tasks
    // - delete temp files
    // - clear large caches
  }
};
```

20.2.6 Rule 4: Avoid rebuilding large widget trees

Widget creation and layout can be expensive. Rebuilding entire sections on every event increases CPU per session and reduces concurrency capacity.
Professional approaches:

- update only changed widgets,
- use model/view for tables so updates are incremental,
- paginate and virtualize large lists,
- avoid deep widget hierarchies when a simpler structure suffices.

20.2.7 Rule 5: Make “idle sessions” cheap

Not every session is actively clicking. The architecture should ensure idle sessions do not:

- hold large memory buffers,
- keep timers running unnecessarily,
- keep long polling activity when not needed,
- keep server-side push enabled without reason.

Design tactics:

- avoid frequent timers per session; use shared scheduling where possible,
- stop background refresh when the page/component is not visible,
- disable live updates for inactive users or idle views.

20.2.8 Rule 6: Control shared resource contention

Even if each session is lightweight, shared bottlenecks can destroy scalability:

- DB connection pool saturation,
- global locks in caches,
- synchronous logging under heavy load,
- filesystem contention for uploads,
- long-running tasks holding worker threads.

Production guidance:

- size connection pools based on concurrency and query latency,
- prefer lock-minimized data structures for shared caches,
- use asynchronous or buffered logging,
- stream large transfers rather than buffering,
- offload heavy tasks to background workers.

20.2.9 Rule 7: Design for horizontal scaling

Scaling beyond a single process requires decisions about session routing.

Sticky sessions (session affinity)

With multiple Wt worker processes (or nodes), a session should usually be routed to the same worker that owns its `WApplication`. This is commonly implemented by:

- a reverse proxy load balancer with session affinity,
- consistent hashing or cookie-based routing,
- a stable mapping from session ID to worker.

Sticky sessions are straightforward but must be paired with:

- health checks (to drain failing nodes),
- graceful restart handling,
- session timeout policies.

Session replication

Replicating full server-side session state across nodes is expensive and complex for widget trees. If you need failover without losing UI state, consider:

- persisting essential state externally (DB/Redis),
- rehydrating views on reconnect using saved view state,
- keeping the live widget tree non-replicated.

Professional posture:

> Treat the widget tree as a *cache of interactive state* and persist only what is required to restore user workflows.

20.2.10 Rule 8: Use backpressure and bounded concurrency

If every session can trigger heavy work, the server will collapse under load. Enforce backpressure:

- cap concurrent expensive tasks per process,
- queue work with bounded queues,
- reject or defer operations when overloaded,
- degrade gracefully (reduced refresh rate, limited exports).

```
class WorkLimiter {
public:
  explicit WorkLimiter(std::size_t max) : max_(max) {}

  bool try_enter() {
    auto cur = active_.load();
    while (cur < max_) {
      if (active_.compare_exchange_weak(cur, cur + 1)) return true;
    }
    return false;
  }

  void leave() { active_.fetch_sub(1); }

private:
  std::atomic<std::size_t> active_{0};
  std::size_t max_{0};
};
```

20.2.11 Rule 9: Treat large tables as paged views, not session-owned datasets

The fastest path to poor scalability is storing entire datasets per session. For tables:

- store only page parameters and selection IDs,
- fetch rows per page from services,

- keep the model small and replace page content on navigation.

```
struct TableQuery {
  int pageIndex{0};
  int pageSize{50};
  std::string search;
  std::string sortKey{"id"};
  bool sortDesc{true};
};
```

20.2.12 Rule 10: Measure session footprint and growth over time

Session scalability requires observability:

- track active sessions, idle sessions, and session churn rate,
- track memory per session (estimated), peak memory, and fragmentation,
- track event latency distribution (p50/p95/p99),
- track DB pool usage and wait times.

Operationally, you want to detect:

- slow leaks (memory increases with session count and does not drop),
- high churn (sessions created/destroyed too often),
- load spikes where latency jumps due to contention.

20.2.13 Practical session scalability patterns in Wt

Pattern: Session view state object

Keep a small state object per session and store only what the UI needs to restore the current view.

```
class MyApp : public Wt::WApplication {
public:
  MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env) {
    state_.pageSize = 50;
  }

private:
  SessionViewState state_;
};
```

Pattern: Shared service singletons (stateless)

- stateless services shared across sessions,
- request-scoped lightweight objects created on demand,
- pooled dependencies (DB connections).

Pattern: Explicit teardown of heavy UI

When switching pages or modes, remove the old subtree instead of keeping it hidden forever.

```
void replaceContent(std::unique_ptr<Wt::WWidget> w) {
  content_->clear();          // remove and destroy old widgets
  content_->addWidget(std::move(w));
}
```

Pattern: Controlled background work

Use background workers for heavy tasks (report generation, imports) and let the session poll for completion or receive a lightweight update, rather than holding a worker thread blocked.

20.2.14 Checklist: session scalability readiness

- Per-session memory is bounded and measured.
- Sessions expire correctly; cleanup reliably releases resources.
- Tables and lists use pagination; no full-dataset retention per session.
- Services are stateless and shared; DB access uses pools.
- Shared bottlenecks are identified (DB pool, locks, logging) and mitigated.
- Expensive operations are limited, queued, or offloaded to background workers.
- Horizontal scaling strategy is defined (sticky sessions plus restore strategy).
- Idle sessions are cheap (few timers, minimal background work).
- Observability exists for sessions, latency, memory, and contention.

20.2.15 Outcome

A session-scalable Wt application can support large numbers of concurrent users without linear memory blow-up, without degrading response times, and without fragile operational behavior. The key is to treat sessions as first-class managed resources: keep them small, keep them bounded, clean them up reliably, and design architecture boundaries that allow horizontal scaling without turning the widget tree into a replicated state burden.

20.3 Threading model

Understanding the threading model of a Wt application is crucial for achieving performance, stability, and scalability. Wt is a server-side C++ framework that handles incoming

HTTP/WebSocket connections, session state, and event dispatch on the server. Internally, Wt uses asynchronous I/O and a multithreaded execution model to maximize throughput while maintaining safety and determinism for application developers. Professional use of Wt requires understanding how threads are used, where concurrency boundaries exist, how to schedule background work, and how to avoid common pitfalls such as race conditions or unsafe access to session-owned state.

20.3.1 Server architecture and concurrency overview

A Wt application is typically run inside a `Wt::WServer` or via the `WRun()` helper function. The server accepts incoming requests and handles them in a way that balances asynchronous I/O with request processing across multiple threads. Wts internal server connector (for example `libwthttp`) is implemented with asynchronous networking and supports multiple threads handling I/O and application work. This allows Wt to handle many simultaneous sessions with responsive event dispatch and minimized blocking on network operations.:contentReference[oaicite:0]index=0
Conceptually, Wts threading model involves:

- **I/O threads**: These threads perform non-blocking accept and read/write operations on sockets. Using asynchronous I/O libraries (often based on `asio` under the hood), I/O threads avoid dedicating one OS thread per connection.
- **Worker threads / thread pool**: When an HTTP request or WebSocket message arrives, the I/O subsystem dispatches work to a pool of worker threads. This pool processes application logic, session state updates, event callbacks, and rendering tasks.
- **Session affinity**: Wt maintains a session per client. Application callbacks (such as signal/slot handlers) that modify the widget tree or session state are always executed on a thread context that is safe for that sessions state. In practice this is scheduled through the servers thread pool with session-affine execution context.

In professional deployments, tuning these thread pools and understanding where blocking work occurs are key for scalability.

20.3.2 Thread safety and Wt application state

Wt applications are inherently concurrent: multiple requests, timers, and server-initiated events (such as Comet or server push) may interleave. However, the widget tree and per-session state must not be accessed concurrently by multiple threads without synchronization.
The general threading safety rules in Wt are:

- **Do not access the same `WApplication` or widget instance concurrently** from multiple threads. All UI-related work must run in the context of the Wt event loop or via safe posting mechanisms.
- **Use `WServer::post()`** when work from an external thread must schedule a callback into a specific sessions context. This ensures that the callback executing widget mutations is performed on the correct session/thread affinity without race conditions.:contentReference[oaicite:1]index=1
- **Avoid long-running work** on the server event threads; offload expensive tasks to dedicated thread pools or asynchronous tasks to prevent starvation of request handling threads.

20.3.3 Event loop and reactor pattern

Internally, Wts networking layer implements a reactor-style event loop. The reactor pattern is a general architecture for scalable I/O whereby a central event demultiplexer waits for socket events and dispatches them to handlers without blocking on individual connections. This allows a small number of I/O threads to manage thousands of concurrent connections efficiently. Tracing the common reactor principles:

- Asynchronous I/O waits for readiness events.
- Event handlers are triggered when a socket is ready for read/write.
- Dispatch to worker threads to process application logic once I/O events are available.

The integration of asynchronous I/O with a thread pool avoids dedicating one thread per socket while keeping worker threads focused on application logic.:contentReference[oaicite:2]index=2

20.3.4 Request handling and worker thread pools

When a new connection arrives, Wt connectors (like `libwthttp` or FastCGI connectors) accept the connection and process the HTTP/WebSocket protocol. The thread that handles the protocol handshake may run in one of the I/O threads. Once the request is parsed and ready, the work is queued for a worker thread that runs the appropriate application logic:

```
// pseudo-code illustrating dispatch
server.onRequest([parsedRequest](auto sessionContext) {
  threadPool.post([parsedRequest, sessionContext] {
    // run application logic safely in worker thread
    handleApplicationRequest(parsedRequest, sessionContext);
  });
});
```

Worker thread pools are configured via Wts server configuration (command-line or XML config) or via programmatic options. Tuning pool size according to hardware and expected load is essential for throughput.

20.3.5 Posting work from non-UI threads

There are occasions when auxiliary threads (for example background jobs, scheduled tasks, or asynchronous I/O callbacks from third-party libraries) need to schedule work that affects a Wt session. Direct widget access from such threads is unsafe. Instead, the correct pattern is:

- Use `WServer::post(sessionId, callback)` to schedule a callback into the targeted sessions context.
- This callback will run in the worker thread that is designated for that sessions event processing.

```
auto server = Wt::WServer::instance();
server->post(sessionId, [] {
    auto app = Wt::WApplication::instance();
    app->root()->addWidget(std::make_unique<Wt::WText>("Update from background"));
});
```

This pattern decouples external thread execution from session-affine operations safely, enforcing correct thread ownership and avoiding undefined behavior.:contentReference[oaicite:3]index=3

20.3.6 Avoiding blocking in event threads

Because worker threads may serve multiple sessions and handle many events, blocking operations (long computations, synchronous I/O, database queries, file system scans) should not be performed directly within Wt thread contexts. Instead:

- Use asynchronous task execution (e.g., task queues, futures) with results posted back via `WServer::post()`.
- Limit blocking to short operations or use dedicated thread pools separate from the core Wt event/work thread pool.

```
std::future<ResultType> backgroundTask = std::async(std::launch::async, [] {
    return doHeavyWork();
});
backgroundTask.then([](std::future<ResultType> result) {
```

```
    Wt::WServer::instance()->post(sessionId, [r = result.get()] {
        // update UI via Wt safely
    });
});
```

This preserves responsiveness of the core event threads.

20.3.7 Thread pool sizing and tuning

Proper sizing of thread pools depends on:

- number of CPU cores (for CPU-bound workloads),
- expected I/O wait times (for I/O-bound workloads),
- anticipated session concurrency,
- and service dependencies (databases, message queues).

A common approach is to start with a pool size equal to the number of hardware threads and adjust based on observed latency, CPU utilization, and queuing times in profiling.

20.3.8 Multi-process and multi-node deployment

While the threading model describes concurrency within a single process, scalable deployments often use multiple processes or nodes. In such scenarios:

- A load balancer with session affinity (sticky sessions) may be required to ensure session continuity.
- Persistent shared state (e.g., caches, databases) must be accessed safely with concurrency control.

- Thread pools and session count per process must be tuned so that each worker process remains responsive and does not exhaust memory or CPU.

The threading model within each process remains consistent; horizontal scaling multiplies capacity rather than changing the fundamental concurrency design.

20.3.9 Common concurrency pitfalls and professional guidance

- **Race conditions**: Avoid shared mutable state across threads. Use locks or atomics where needed, and prefer immutable data for cross-thread communication.
- **Deadlocks**: Keep lock scopes minimal; avoid holding locks when posting back to Wt threads.
- **UI access from wrong threads**: Always use `WServer::post()` or equivalent safe dispatch.
- **Blocking event threads**: Offload heavy or blocking work to separate thread pools.

20.3.10 Outcome

A correct understanding of Wts threading model enables professional performance engineering. By separating asynchronous I/O, worker threads, and session-affine event handling, applications can scale to hundreds or thousands of concurrent sessions, maintain responsiveness under load, and avoid common concurrency bugs. Thoughtful use of posting mechanisms, thread pools, and background task integration ensures that Wt applications remain robust, maintainable, and performant under real-world workloads.

20.4 Response optimization

Response optimization in Wt is the discipline of minimizing the time and bytes required to deliver a correct UI update or HTTP response to the browser, while preserving security, correctness, and maintainability. Because Wt is server-side and event-driven, the user experience depends on:

- server latency (CPU time + lock contention + database time),
- network latency and bandwidth,
- response size and serialization overhead,
- the number of round-trips required for a user action,
- and how efficiently UI diffs are produced and shipped.

Professional response optimization is not about micro-optimizing a single line; it is a systematic approach that includes protocol behavior, caching, compression, payload shaping, streaming, and the elimination of unnecessary work.

20.4.1 What “response” means in Wt

Depending on the situation, a Wt “response” may be:

- the initial HTML/CSS/JS bootstrap delivered at page load,
- a partial update triggered by a signal/slot event (server-side diff),
- an API-like HTTP handler response (JSON, file download),
- a server push update (when enabled) delivered asynchronously.

Therefore, response optimization in Wt must consider both:

- **payload optimization**: reducing bytes and serialization cost,
- **latency optimization**: reducing server time, round-trips, and blocking.

20.4.2 Principle 1: Optimize the critical path (latency before throughput)

The user perceives the time from click to visible update. Optimize the critical path:

- avoid synchronous heavy work inside UI callbacks,
- avoid calling multiple services sequentially if they can be combined,
- reduce database round-trips and avoid N+1 queries,
- limit lock contention (shared caches, global mutexes),
- return early for validation failures before doing expensive work.

```
submit_->clicked().connect([this] {
  // Validate first (cheap), fail fast:
  if (!model_->validate()) {
    showValidationErrors();
    return;
  }

  // Expensive operations only after validated input:
  auto r = service_.submit(use_case_request());
  handleResult(r);
});
```

20.4.3 Principle 2: Reduce response size by design

Response size affects:

- network transfer time,
- browser parsing time (initial bootstrap),
- server serialization and diff generation time.

Professional methods:

- paginate large datasets; never send full tables when a page is enough,
- send only fields needed for a view (avoid “select *”),
- avoid rendering hidden heavy widgets that are not visible,
- avoid embedding large binary content in HTML responses.

20.4.4 Principle 3: Exploit incremental UI updates (diff-based thinking)

A major advantage of Wt is the ability to update only what changed. To benefit:

- update single widgets, not whole containers,
- avoid rebuilding large widget subtrees on small changes,
- use model/view patterns for tables to update page data efficiently,
- keep widget trees stable so Wt can compute minimal diffs.

```
void updateStatus(const std::string& msg) {
  statusText_->setText(Wt::WString::fromUTF8(msg)); // minimal UI update
}
```

20.4.5 Principle 4: Cache and reuse what is stable

Caching is response optimization when used correctly. Cache:

- computed values that are expensive and reused,
- reference data (roles, statuses),
- templates or prebuilt fragments if applicable,
- authorization results if safely scoped and short-lived.

Professional caching must be:

- bounded (size/time limits),
- invalidated correctly,
- safe under multi-tenant constraints,
- observable (hit/miss metrics).

```
struct CacheEntry {
  std::string value;
  std::chrono::steady_clock::time_point expiresAt;
};
```

20.4.6 Principle 5: Use compression strategically

Compression reduces payload size but costs CPU. Use it where it has the best payoff:

- compress text payloads (HTML, CSS, JS, JSON),
- avoid compressing already-compressed binaries (JPEG, PNG, PDF) unless policy requires it,

- ensure compression is enabled in your reverse proxy or connector configuration where appropriate.

In production deployments, compression is commonly handled at the reverse proxy layer, which can apply consistent policy and leverage optimized implementations.

20.4.7 Principle 6: Minimize round-trips

Each interaction that requires multiple server round-trips increases perceived latency. Reduce round-trips by:

- validating locally where safe (basic constraints) while still validating authoritatively on the server,
- combining dependent server operations into one service call,
- using batched updates rather than multiple sequential UI-triggered requests,
- avoiding chatty per-field server validations for large forms.

20.4.8 Principle 7: Streaming for large responses

Large downloads, exports, and report generation must not buffer entire payloads in memory. Stream:

- file downloads,
- CSV/JSON exports,
- generated reports.

Streaming reduces peak memory, enables early delivery, and avoids long GC-like pauses from large allocations.

```
void streamFile(const std::string& path, Wt::Http::Response& resp) {
  resp.setMimeType("application/octet-stream");
  resp.addHeader("Content-Disposition", "attachment; filename=\"file.bin\"");
  // In practice, read in chunks and write to resp.out().
}
```

20.4.9 Principle 8: Optimize database interaction for response time

For most Wt applications, the database dominates response time. Optimize DB interaction:

- use prepared statements where supported,
- avoid N+1 patterns (fetch related data in one query or in batches),
- apply proper indexes aligned with sorting and filtering,
- limit result sets strictly (pagination),
- avoid expensive counts on every page unless required.

20.4.10 Principle 9: Control logging overhead in hot paths

Logging can silently become a performance bottleneck:

- synchronous disk flush per request,
- formatting large log strings,
- high-volume debug logs in production.

Professional approach:

- use leveled logging,
- avoid expensive formatting unless the level is enabled,

- batch or async logging where possible,
- log structured, minimal fields in hot paths.

```
if (logger_.isDebugEnabled()) {
  logger_.debug("User id=" + std::to_string(userId) + " clicked submit");
}
```

20.4.11 Principle 10: Avoid accidental response bloat from UI state

Server-side UI systems can bloat responses if:

- too many widgets are created and retained,
- hidden containers still contain heavy content,
- frequent full refreshes are triggered instead of targeted updates.

Professional guidance:

- remove unused widgets (`clear()` or `removeChild()`),
- lazy-load tabs and heavy sections only when needed,
- paginate and virtualize large lists.

20.4.12 Practical patterns for Wt response optimization

Pattern: “Load state” UI component

Always give feedback during server work to prevent duplicate clicks and reduce perceived latency.

```
void setLoading(bool on) {
  submit_->setDisabled(on);
  spinner_->setHidden(!on);
}
```

Pattern: Bounded DTOs for table pages

Return only what the table needs.

```
struct RowDTO {
  std::int64_t id{};
  std::string name;
  std::string status;
};

template <class Row>
struct PageResult {
  std::vector<Row> rows;
  int pageIndex{0};
  int pageSize{50};
  bool hasNext{false};
};
```

Pattern: Cache reference lookups

Statuses and roles rarely change and can be cached safely with TTL.

20.4.13 Measurement: optimization without metrics is guessing

Professional response optimization requires measurement:

- record request latency (p50, p95, p99),
- record DB time separately from application time,
- record response sizes for major endpoints/events,
- monitor thread pool queue time and saturation,
- monitor session memory and growth.

A helpful mental model is to split response time into:

- **queue time** (waiting for a worker),
- **compute time** (CPU, rendering diffs),
- **I/O time** (DB, storage, network).

20.4.14 Checklist: response optimization readiness

- UI updates are incremental; large subtrees are not rebuilt unnecessarily.
- All tables are paginated; page size is bounded.
- Sorting and filtering are server-side and index-aligned.
- DB queries are optimized; no N+1 patterns in table views.
- Large downloads/exports are streamed, not buffered.
- Caching is bounded and correctly invalidated.
- Compression is enabled where it helps and not wasted on already-compressed payloads.
- Round-trips are minimized; operations are batched where practical.
- Logging overhead is controlled in hot paths.
- Latency and response sizes are measured and monitored.

20.4.15 Outcome

When response optimization is engineered correctly, Wt applications maintain low latency under load, deliver responsive server-side UX, reduce bandwidth usage, and preserve stability. The application remains scalable because work is bounded, incremental, and measurable, and performance decisions are driven by observed bottlenecks rather than assumptions.

Security in Wt Applications

21.1 Session security

Session security is the foundation of application security. If an attacker can steal, fixate, forge, or reuse a session, they effectively bypass authentication and authorization controls. In Wt applications, session security has an additional architectural dimension: a session is not just an authentication token, it is commonly the server-side anchor for a `Wt::WApplication` instance and its widget tree. This makes session compromise both a security problem and a stability problem.

This section provides a production-grade approach to session security in Wt applications: secure transport, cookie hardening, fixation prevention, CSRF defenses, idle timeouts, logout semantics, multi-process considerations, and auditing.

21.1.1 Threat model for sessions

A professional design explicitly defends against:

- **Session hijacking**: attacker obtains a valid session identifier (XSS, malware, packet capture, insecure cookies).
- **Session fixation**: attacker forces a victim to authenticate using a session ID chosen or known by the attacker.

- **Session replay**: attacker reuses a captured token, especially when tokens are long-lived and not bound to context.
- **Cross-site request forgery (CSRF)**: attacker causes a victims browser to send authenticated requests without consent.
- **Weak logout semantics**: session remains valid after logout, allowing reuse.
- **Insecure session transport**: missing TLS, mixed content, or proxies misconfigured.
- **Overly long idle lifetimes**: abandoned sessions remain valid indefinitely.
- **Misrouting in clusters**: a session is routed to the wrong process/node and falls back to unsafe patterns.

21.1.2 Core principle: session security is layered

No single measure is sufficient. A secure session model is built from layers:

- Transport security (TLS, HSTS).
- Secure session identifier generation and rotation.
- Cookie hardening (Secure, HttpOnly, SameSite).
- CSRF protections for state-changing operations.
- Idle timeout and absolute lifetime controls.
- Explicit logout invalidation.
- Server-side authorization checks on every action (session is not authorization).
- Monitoring and audit logging.

21.1.3 Transport security: TLS is mandatory

Sessions must never travel over plaintext HTTP. Professional requirements:

- **TLS everywhere**: all pages and endpoints, including login, logout, assets, uploads, and websockets.
- **Redirect HTTP to HTTPS**: do not allow authenticated traffic over HTTP.
- **Avoid mixed content**: a single HTTP resource on an HTTPS page can degrade security.
- **Prefer HSTS**: force browsers to use HTTPS for your domain.

In production, TLS is typically terminated at a reverse proxy. Ensure the application can accurately detect secure transport when behind a proxy (correct forwarded headers and trusted proxy configuration).

21.1.4 Session identifier properties

A session identifier must be:

- **Unpredictable**: generated from a cryptographically secure random source.
- **Sufficiently long**: high entropy to resist guessing.
- **Opaque**: no embedded user ID, timestamps, or structure that leaks information.
- **Single-purpose**: do not reuse tokens across different security contexts.

Never design your own “simple token” format. Use established, cryptographically strong session ID generation.

21.1.5 Cookie hardening

For browser-based sessions, cookies are the primary carrier. Harden them:

- **Secure**: cookie is sent only over HTTPS.
- **HttpOnly**: cookie cannot be read by JavaScript (reduces impact of XSS on token theft).
- **SameSite**: reduces CSRF risk by controlling cross-site cookie sending behavior.
- **Path/Domain scoping**: minimize exposure; restrict the cookie to the needed path and host scope.

Practical policy

- **SameSite=Lax** is a strong default for many traditional apps.
- **SameSite=Strict** provides stronger CSRF resistance but can break legitimate cross-site flows.
- **SameSite=None** requires **Secure** and should be used only when you truly need third-party contexts.

21.1.6 Session fixation prevention: rotate session ID on privilege change

Session fixation is prevented by rotating the session identifier whenever privilege changes:

- after successful login,
- after privilege elevation (admin mode, role changes),
- after switching accounts,
- after enabling sensitive features.

Golden rule

> A user must never authenticate into a session identifier that existed before authentication.

Implementation patterns:

- invalidate the old session server-side,
- issue a new session ID,
- re-bind only minimal necessary state to the new session (avoid copying unbounded state).

21.1.7 CSRF protection in a session-based system

Even with SameSite cookies, CSRF defenses must exist for state-changing requests:

- **Synchronizer token**: a per-session or per-request CSRF token included in forms and validated server-side.
- **Double-submit cookie** (advanced): token in cookie plus token in request body/header.
- **Strict method semantics**: only allow state changes via POST/PUT/PATCH/DELETE; never via GET.

Wt form handling is server-driven, but the security rule remains: every state change must be intentional and protected.

21.1.8 Idle timeout and absolute lifetime

Session lifetime must be bounded:

- **Idle timeout**: expire the session after inactivity (e.g., 15–60 minutes depending on sensitivity).

- **Absolute lifetime**: expire the session after a maximum total lifetime even if active (reduces replay window).
- **Sliding renewal** (optional): renew within safe bounds for usability.

Additionally:

- treat “remember me” as a different security primitive (a refresh token), not the same as a normal session cookie,
- require re-authentication for sensitive actions (password change, account deletion, role changes).

21.1.9 Logout semantics: invalidate server-side, not only client-side

A secure logout must:

- invalidate the session server-side (mark token unusable),
- clear cookies client-side,
- revoke long-lived tokens (remember-me) if applicable.

Clearing cookies alone is insufficient if the token remains valid and can be replayed.

21.1.10 Session binding and anomaly detection (defense-in-depth)

For high-security applications, consider binding sessions to additional context:

- detect impossible travel patterns,
- monitor user-agent and device fingerprint changes (carefully: do not lock out legitimate users),

- optionally bind to a coarse IP range (beware mobile networks and proxies),
- rate-limit sensitive endpoints (login, token refresh, password reset).

Treat binding as a signal, not an absolute rule, unless your environment is tightly controlled.

21.1.11 Cluster and multi-process considerations

Wt sessions commonly have server-side state. In horizontal scaling:

- **Sticky sessions (session affinity)** is typically required so a user returns to the process that owns their live session state.
- If a process restarts, sessions may be lost; design for safe reconnection:
  - persist essential workflow state externally (DB/cache),
  - treat the widget tree as reconstructible state,
  - re-authenticate or revalidate tokens on reconnect.

Never implement "fallback" behavior that weakens session integrity to avoid session loss. Security must be preserved even during failover.

21.1.12 Professional patterns in C++ for secure session primitives

Pattern: CSRF token generation

Generate CSRF tokens using cryptographically secure randomness and store server-side. Expose only the token value to the form.

```
#include <array>
#include <cstdint>
#include <string>
```

```
// This is a conceptual example: use a cryptographically secure RNG.
// On Linux, prefer getrandom() or a well-audited crypto library.
// On Windows, use BCryptGenRandom or equivalent.

static std::string to_hex(const std::uint8_t* data, std::size_t n) {
  static const char* k = "0123456789abcdef";
  std::string out;
  out.resize(n * 2);
  for (std::size_t i = 0; i < n; ++i) {
    out[2*i]   = k[(data[i] >> 4) & 0xF];
    out[2*i+1] = k[(data[i]     ) & 0xF];
  }
  return out;
}

struct CsrfToken {
  std::string value; // hex random
};

CsrfToken generate_csrf_token(/*secure_rng*/) {
  std::array<std::uint8_t, 32> bytes{}; // 256-bit token
  // secure_rng.fill(bytes.data(), bytes.size()); // required
  return CsrfToken{ to_hex(bytes.data(), bytes.size()) };
}
```

Pattern: constant-time comparison

Compare secrets without leaking timing information:

```
#include <string>
#include <cstdint>

static bool constant_time_equal(const std::string& a, const std::string& b) {
```

```
  if (a.size() != b.size()) return false;
  std::uint8_t diff = 0;
  for (std::size_t i = 0; i < a.size(); ++i) {
    diff |= static_cast<std::uint8_t>(a[i] ^ b[i]);
  }
  return diff == 0;
}
```

Pattern: session token rotation protocol (conceptual)

On successful login:

- invalidate old session token,
- create a new session token,
- set cookie with Secure/HttpOnly/SameSite,
- bind user identity to the new session server-side.

```
// Conceptual flow; concrete APIs depend on your session store and HTTP integration.
struct SessionStore {
  void invalidate(const std::string& sessionId);
  std::string create_new_session_for_user(std::int64_t userId);
};

void on_login_success(SessionStore& store,
                      const std::string& oldSessionId,
                      std::int64_t userId) {
  store.invalidate(oldSessionId);
  const auto newSessionId = store.create_new_session_for_user(userId);

  // Set-Cookie should include: Secure; HttpOnly; SameSite=Lax/Strict; Path=/
  // Apply max-age based on policy; keep it short for normal sessions.
```

```
  // response.addCookie(... newSessionId ...);
}
```

21.1.13 Operational security: headers and policies

Even though Wt is server-side, browser security headers still matter:

- **Content Security Policy (CSP)** to reduce XSS impact.
- **X-Content-Type-Options: nosniff** to reduce content-type confusion.
- **Frame-ancestors / X-Frame-Options** to reduce clickjacking.
- **Referrer-Policy** to reduce token leakage in referrers.

Apply these at the reverse proxy or within response generation consistently across the application.

21.1.14 Auditing and monitoring

Security is incomplete without visibility. Log (carefully):

- login success/failure (without storing passwords or secrets),
- session creation and rotation events,
- logout and token revocation,
- suspicious activity (multiple failed logins, unusual download spikes),
- CSRF validation failures.

Never log session IDs or CSRF tokens in plaintext. If you must correlate, log a hashed/truncated representation.

21.1.15 Checklist: secure sessions in Wt

- TLS enforced everywhere; HTTP redirected to HTTPS.
- Session IDs are cryptographically strong and opaque.
- Cookies use Secure + HttpOnly + appropriate SameSite.
- Session ID rotates on login and privilege changes (fixation prevention).
- CSRF protection exists for all state-changing operations.
- Idle timeout and absolute lifetime are enforced.
- Logout invalidates server-side session and clears client cookies.
- “Remember me” is implemented as a separate, revocable long-lived token.
- Cluster deployment uses safe session routing (sticky sessions) without weakening integrity.
- Security headers and monitoring are enabled; secrets are not logged.

21.1.16 Outcome

A professionally secured session model ensures that authentication remains meaningful, that user actions cannot be forged via CSRF, and that stolen tokens have limited usefulness. In Wt, where sessions may anchor server-side UI state, strong session security also improves stability and operational safety by preventing hostile or malformed session usage from becoming an application-wide risk.

21.2 Password hashing

Password hashing is a foundational element of application security. When users authenticate with a password, the server must never store or compare plaintext passwords. Instead, passwords must be transformed via a cryptographically strong one-way function that resists brute-force, dictionary attacks, and adaptive cracking attempts. In production systemsincluding Wt applicationspassword hashing must adhere to modern standards using memory-hard algorithms with adjustable computational cost.
This section presents professional practices for password hashing, appropriate algorithms, safety considerations, storage formats, verification flows, migration strategies, and C++ implementation patterns.

21.2.1 Why hashing; not encryption or plain storage

Storing plaintext passwords or reversible encryption is unacceptable because:

- server compromise would expose all user credentials,
- encryption keys themselves become a single point of failure,
- hashed passwords limit the usefulness of exfiltrated data.

A secure hash must be:

- computationally slow enough to deter brute-force,
- memory-hard to resist GPU/ASIC optimization,
- salted uniquely per password to prevent precomputation attacks,
- parameterizable to increase cost over time.

21.2.2 Professional algorithms: Argon2 and bcrypt

Modern password hashing standards emphasize memory hardness.

Argon2 (recommended) Argon2 is the current winner of the Password Hashing Competition and is widely accepted as the state-of-the-art. It has three variants (Argon2i, Argon2d, Argon2id), with Argon2id recommended for general password hashing due to resistance to both side-channel and GPU attacks. Parameters include:

- `time_cost`: number of iterations,
- `memory_cost`: memory usage in kibibytes,
- `parallelism`: degree of parallelism.

bcrypt (well-established alternative) bcrypt is older than Argon2 but remains acceptable. It uses the Eksblowfish keying algorithm with configurable `cost` (log2 rounds). It is not as memory-hard as Argon2 but still slows brute force.

21.2.3 Salting and parameterization

Every hashed password must include a unique salt. A salt is a random value (e.g., 16 bytes or more) appended or prepended to the plaintext before hashing. Salts prevent attackers from using precomputed rainbow tables.
Hash outputs should bundle:

- algorithm identifier,
- parameters (cost/memory/time),
- salt,

- hash digest.

This enables verifying without external parameter storage.

21.2.4 Hash storage format

Hash outputs are stored in a single text field in the database. A canonical storage encoding includes algorithm, parameters, salt, and digest using a colon or dollar-separated format. Example Argon2 format:

```
$argon2id$v=19$m=65536,t=3,p=4$<base64salt>$<base64hash>
```

21.2.5 Verification flow

At login:

1. receive user-supplied password,
2. retrieve stored hash entry,
3. parse algorithm and parameters,
4. re-compute hash on input with same salt and parameters,
5. compare in constant time.

Comparison must be constant-time to avoid timing attacks.

21.2.6 Parameter selection and cost tuning

Choose parameters based on:

- server CPU performance,

- available memory,
- authentication throughput targets,
- expected user base concurrency.

Do not hardcode parameters; make them configurable to increase cost over time as hardware improves.
Example Argon2 reasonable starting point (2025+):

- memory cost: 64 MiB256 MiB,
- time cost: 24 iterations,
- parallelism: number of CPU cores or 14.

21.2.7 Migration and algorithm agility

Over time, preferred algorithms evolve. Systems must:

- detect outdated hash formats (e.g., old bcrypt with low cost),
- re-hash passwords on next login with updated parameters,
- store version metadata in the hash string.

This enables algorithm agility without forcing password resets.

21.2.8 C++ integration patterns

Production C++ applications typically use well-audited libraries for hashing. Rolling your own is unsafe.

Argon2 example using an external library This example shows the conceptual usage of Argon2 via a hypothetical C++ wrapper:

```
// Pseudocode: actual implementation depends on the chosen library.
#include <string>
#include <vector>

std::string hashPasswordArgon2(const std::string& password) {
  // parameters (example):
  uint32_t timeCost   = 3;
  uint32_t memCost    = 1 << 16; // 64 MiB
  uint32_t parallel   = 4;
  size_t saltLen      = 16;
  std::vector<uint8_t> salt(saltLen);
  secureRandom(salt.data(), saltLen);

  // argon2Hash encapsulates algorithm, parameter encoding, base64, etc.
  return argon2Hash(password, salt, timeCost, memCost, parallel);
}

bool verifyPasswordArgon2(const std::string& password,
                          const std::string& storedHash) {
  return argon2Verify(storedHash, password);
}
```

bcrypt example using a C++ wrapper

```
#include <string>

std::string hashPasswordBcrypt(const std::string& password, int cost) {
  return bcryptHash(password, cost); // wrapper provided by library
}
```

```
bool verifyPasswordBcrypt(const std::string& password,
                          const std::string& storedHash) {
  return bcryptVerify(storedHash, password);
}
```

Always choose well-maintained libraries with active maintenance and security reviews.

21.2.9 Constant-time comparison

When comparing digests:

```
#include <string>
#include <cstdint>

bool constantTimeEqual(const std::string& a, const std::string& b) {
  if (a.size() != b.size()) return false;
  uint8_t diff = 0;
  for (size_t i = 0; i < a.size(); ++i) {
    diff |= static_cast<uint8_t>(a[i] ^ b[i]);
  }
  return diff == 0;
}
```

Never use == for password hashes.

21.2.10 Secure random for salt and secrets

Salt and any cryptographic nonce must be produced by a secure RNG:

- on Linux, use `getrandom()` or `/dev/urandom`,
- on Windows, use `BCryptGenRandom()`,
- or rely on a vetted crypto library.

21.2.11 Protecting against offline attacks

Even with strong hashing, if an attacker obtains the password database, offline brute-force is still possible. Strong algorithms and high cost parameters increase attacker compute cost. Additional protections:

- password complexity policies (length and entropy),
- account lockouts or exponential delays after repeated failures,
- multi-factor authentication (MFA) to reduce reliance on passwords alone.

21.2.12 Avoiding anti-patterns

Do not:

- use general hash functions (MD5, SHA-1, SHA-256) directly without key stretchingthese are too fast for password hashing.
- store passwords encrypted with a symmetric key.
- reuse the same salt for multiple users.
- transmit plaintext passwords over insecure channels (enforce TLS).

21.2.13 Testing and validation

Include tests that verify:

- hashing and verification succeed for valid credentials,
- verification fails for incorrect passwords,
- rehash triggers on outdated cost parameters,

- no exceptions or leaks occur during hashing operations.

```
#include <cassert>

void test_hash_and_verify() {
  auto hash = hashPasswordArgon2("CorrectHorseBatteryStaple");
  assert(verifyPasswordArgon2("CorrectHorseBatteryStaple", hash));
  assert(!verifyPasswordArgon2("WrongPassword", hash));
}
```

21.2.14 Migration strategies

To migrate from an old algorithm:

- accept old hashes temporarily,
- on successful login, rehash with the new algorithm and update storage,
- mark the account as upgraded to avoid repeated migration on every login,
- remove support for weak algorithms once all accounts are upgraded.

21.2.15 Outcome

By following professional password hashing practices, Wt applications:

- protect user credentials against server compromise,
- resist offline brute-force and GPU/ASIC attacks,
- remain adaptable as hardware and algorithms evolve,
- and provide a secure foundation for authentication systems.

21.3 HTTPS concepts

HTTPS is HTTP carried over Transport Layer Security (TLS). In a production Wt application, HTTPS is not merely a "nice-to-have" encryption layer; it is the foundation that protects:

- credentials during login,
- session identifiers (cookies and tokens),
- private user data in transit,
- form submissions (including file uploads),
- and the integrity of the delivered UI (preventing in-transit modification).

This section explains HTTPS and TLS concepts that matter operationally and architecturally for Wt applications, including certificate trust, handshake behavior, modern protocol features, proxy termination, and the security headers that complete an HTTPS deployment.

21.3.1 HTTPS versus HTTP: what changes

Without TLS, HTTP traffic can be observed and modified by any party on the network path (local networks, ISP infrastructure, proxies, compromised routers). HTTPS introduces:

- **Confidentiality**: encrypts request/response bytes (protects against passive eavesdropping).
- **Integrity**: detects tampering (protects against active modification).
- **Authentication**: verifies the server identity via certificates (protects against impersonation).

In practice, HTTPS prevents:

- credential theft by in-transit interception,
- session hijacking by cookie capture,
- injection of malicious scripts into your pages,
- modification of form actions or API endpoints.

21.3.2 TLS building blocks

Certificates and trust chains

A TLS server certificate binds a domain name to a public key. Browsers accept that binding only if:

- the certificate is issued by a Certificate Authority (CA) trusted by the browser/OS,
- the certificate is valid (time window, signature),
- the certificate matches the requested hostname,
- and the certificate chain is complete and verifiable.

The browser validates the chain:

- server certificate $\rightarrow$ intermediate CA certificate(s) $\rightarrow$ trusted root CA.

Operational consequence:

- your server must present the correct certificate and intermediate chain,
- certificate renewal must be automated,
- private keys must be protected (file permissions, HSM/secure storage where applicable).

TLS handshake (conceptual)

A modern TLS handshake establishes:

- agreement on protocol version and cipher suite,
- key exchange to derive session keys,
- authentication of the server certificate,
- and encrypted application traffic keys.

Key points for application engineers:

- TLS negotiation happens *before* HTTP is exchanged.
- Once established, TLS provides an encrypted pipe; HTTP semantics remain the same, but are protected.
- The handshake cost affects latency; modern TLS configurations aim to minimize it (e.g., TLS 1.3).

TLS 1.3 and forward secrecy

Modern deployments prefer TLS 1.3 (and TLS 1.2 only when compatibility requires it). A critical property is **forward secrecy**:

- compromise of the server private key in the future does not allow decryption of *past* recorded sessions.

Forward secrecy depends on ephemeral key exchange (e.g., ECDHE). Production policy should explicitly avoid legacy, non-forward-secret key exchange.

21.3.3 HTTPS and Wt: typical deployment topologies

Topology A: TLS terminated at a reverse proxy

This is the most common production layout:

- browser connects via HTTPS to a reverse proxy (Nginx/Apache/HAProxy),
- proxy terminates TLS,
- proxy forwards plain HTTP to the Wt server on an internal network interface,
- proxy applies security headers, compression, and caching policies.

Benefits:

- centralized TLS configuration and certificate automation,
- simpler Wt connector configuration,
- consistent security headers,
- efficient HTTP/2 and (optionally) HTTP/3 at the edge.

Critical requirement:

- the internal hop (proxy $\rightarrow$ Wt) must be trusted and protected (localhost, private VLAN, firewall).

Topology B: TLS terminated inside the Wt connector

This is possible but less common in large systems, because certificate automation, header policy, and multi-service routing are typically better managed at the proxy layer.

21.3.4 Reverse proxy correctness: the “real scheme” problem

When TLS is terminated at a proxy, the Wt server receives HTTP internally and may believe the scheme is `http`. If the application:

- generates absolute URLs,
- sets cookie security attributes,
- performs redirects,
- or enforces security policies based on scheme,

it must correctly determine that the external scheme is `https`.
Professional solution:

- configure the proxy to send a trusted header indicating the original scheme (e.g., `X-Forwarded-Proto: https`),
- configure the application to trust that header only from known proxy addresses.

Never trust forwarded headers from the public internet unless the request is known to come from your proxy.

21.3.5 HSTS: enforcing HTTPS at the browser

HTTP Strict Transport Security (HSTS) is a response header that tells the browser:

> “For a period of time, do not use HTTP for this host; always use HTTPS.”

Example policy (strict production posture):

```
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
```

Operational considerations:

- Start with a shorter `max-age` during rollout, then increase once confirmed.
- Do not enable `includeSubDomains` unless all subdomains can serve HTTPS safely.
- Preload is a strong commitment; removing it is slow and operationally risky.

21.3.6 Cookies and HTTPS: session security coupling

Session cookies must be hardened; HTTPS is required but not sufficient.
Recommended cookie attributes:

- **Secure**: cookie is sent only over HTTPS.
- **HttpOnly**: cookie is not readable by JavaScript (reduces token theft via XSS).
- **SameSite**: reduces CSRF by limiting cross-site cookie sending.

Important relationship:

- If HTTPS is not enforced everywhere, cookies can leak or be downgraded.
- HSTS and HTTP→HTTPS redirects reduce downgrade opportunities.

21.3.7 TLS configuration goals: security and performance

A production TLS configuration aims to:

- enable modern protocol versions (prefer TLS 1.3),
- disable legacy protocols and weak ciphers,
- enable forward secrecy,
- reduce handshake overhead,

- support HTTP/2 where beneficial.

From a performance perspective:

- TLS adds CPU overhead, but modern ciphers are optimized and typically acceptable.
- The dominant cost is often handshake latency; keep connections reused and avoid needless reconnects.

21.3.8 OCSP and revocation: practical view

Certificate revocation exists to invalidate compromised or misissued certificates. In real systems:

- revocation checks may not always be strictly enforced by all clients,
- but operational hygiene still matters: protect keys, rotate on compromise, automate renewals.

Some deployments use OCSP stapling at the proxy to improve revocation signaling and reduce client-side latency. Treat this as defense-in-depth, not a substitute for key management.

21.3.9 Certificate lifecycle and automation

Operational reality: certificates expire and must be renewed reliably. A professional HTTPS posture requires:

- automated renewal using an ACME client,
- monitoring for renewal failures,
- safe reload of proxy configuration on renewal,
- readiness for shorter certificate lifetimes over time (automation is not optional).

Never rely on manual renewal in production.

21.3.10 Mixed content, origin integrity, and application correctness

Even with HTTPS, your security can be undermined if:

- the page loads scripts or assets over HTTP (mixed content),
- external dependencies are not controlled and can change unexpectedly,
- your application allows user-supplied HTML/JS injection (XSS).

For Wt applications:

- ensure all static assets are delivered over HTTPS,
- set strict content security policies where possible,
- avoid rendering untrusted HTML; prefer safe text APIs.

21.3.11 WebSockets and HTTPS: WSS

If your Wt configuration uses WebSockets for interactive updates:

- HTTPS corresponds to **WSS** (WebSocket over TLS),
- the proxy must allow WebSocket upgrade headers,
- session security assumptions still apply (cookies, CSRF model, origin checks).

21.3.12 Practical edge configuration example (reverse proxy)

A minimal, security-oriented proxy sketch (conceptual):

```
# Redirect HTTP to HTTPS
server {
  listen 80;
```

```
  server_name example.com;
  return 301 https://$host$request_uri;
}

# HTTPS termination + proxy to Wt
server {
  listen 443 ssl http2;
  server_name example.com;

  ssl_protocols TLSv1.2 TLSv1.3;

  # Certificates (paths depend on your ACME client)
  ssl_certificate     /etc/ssl/example/fullchain.pem;
  ssl_certificate_key /etc/ssl/example/privkey.pem;

  # HSTS (enable after confirming HTTPS works for all subdomains/policies)
  add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;

  # Hardening headers (examples; align with your app needs)
  add_header X-Content-Type-Options "nosniff" always;
  add_header Referrer-Policy "strict-origin-when-cross-origin" always;

  location / {
    proxy_pass http://127.0.0.1:8080;

    # Forward original scheme/host information
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    # WebSocket support (if used)
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
```

```
    proxy_set_header Connection "upgrade";
  }
}
```

This example focuses on the essential concepts:

- force HTTPS,
- terminate TLS at the edge,
- forward scheme/host correctly,
- apply HSTS cautiously,
- allow WebSocket upgrades when needed.

21.3.13 Common mistakes to avoid

- Allowing HTTP for authenticated pages (session ID exposure risk).
- Setting cookies without `Secure` when HTTPS is required.
- Trusting `X-Forwarded-*` headers from untrusted sources.
- Enabling HSTS `includeSubDomains` without HTTPS coverage for all subdomains.
- Shipping mixed content (HTTP assets on HTTPS pages).
- Relying on manual certificate renewal and lacking monitoring.
- Treating TLS as “done” and neglecting key management and rotation procedures.

21.3.14 Checklist: HTTPS readiness for Wt applications

- HTTPS enforced for all endpoints; HTTP redirects to HTTPS.
- TLS configuration prefers modern versions and forward secrecy.
- Certificates are renewed automatically and monitored.
- HSTS is deployed with an intentional rollout plan.
- Session cookies use Secure + HttpOnly + appropriate SameSite.
- Reverse proxy forwards scheme/host correctly and only from trusted proxies.
- WebSockets (if enabled) run over WSS and upgrade headers are configured.
- Mixed content is eliminated; security headers are consistent.

21.3.15 Outcome

When HTTPS is engineered correctly, it protects credentials and session integrity, prevents in-transit tampering, and provides a secure transport foundation for all Wt interactions. In production-grade Wt deployments, HTTPS is treated as a complete system: TLS termination, certificate lifecycle automation, secure headers, correct proxy signaling, and browser-enforced policies that together prevent downgrade and interception attacks.

21.4 Critical security pitfalls

A secure Wt application is not achieved by a single security feature. Most production breaches occur because of overlooked *pitfalls*: small design shortcuts, incorrect assumptions about trust boundaries, or missing operational controls. Wts server-side UI model reduces certain browser-side risks, but it does not eliminate classic web threats. In fact, Wt introduces its own

high-impact failure modes because sessions and widget trees can be long-lived and stateful on the server.
This section lists the most critical pitfalls that repeatedly appear in real-world systems, explains why they are dangerous, and provides defensive engineering patterns suitable for production-grade Wt applications.

21.4.1 Pitfall 1: Treating TLS as optional or inconsistent

Failure mode: Allowing any authenticated endpoint over HTTP (even temporarily) exposes session cookies and credentials to interception and tampering.
Consequences:

- session hijacking (stolen cookie),
- credential theft,
- injected scripts and modified forms.

Professional defense:

- enforce HTTPS everywhere,
- redirect HTTP to HTTPS at the edge,
- deploy HSTS with a controlled rollout plan,
- never serve mixed content.

21.4.2 Pitfall 2: Weak session management (fixation, hijacking, long lifetimes)

Failure mode: Not rotating session IDs at login, storing sessions too long, or using insecure cookie attributes.

Consequences:

- session fixation (victim authenticates into attacker-known session),
- replay of stolen session ID,
- persistent compromise due to long-lived sessions.

Professional defense:

- rotate session ID on login and privilege changes,
- use cookie attributes Secure + HttpOnly + appropriate SameSite,
- enforce idle timeout and absolute lifetime,
- invalidate sessions server-side on logout.

21.4.3 Pitfall 3: Relying on "server-side UI" as an XSS shield

Failure mode: Assuming Wt automatically eliminates XSS because it renders widgets server-side.
Wt reduces XSS risk when you consistently render untrusted data as text. However, XSS remains possible if:

- you insert raw HTML,
- you render user content as markup,
- you build JavaScript strings from untrusted data,
- you expose unsafe custom resources.

Professional defense:

- default to safe text rendering APIs,
- never concatenate untrusted values into script or HTML,
- sanitize/escape explicitly when markup is required,
- deploy Content Security Policy (CSP) where feasible.

21.4.4 Pitfall 4: Missing CSRF protection on state changes

Failure mode: Any endpoint that changes state (update profile, submit order, delete record) can be triggered cross-site by an attacker if CSRF defenses are missing.

Professional defense:

- ensure state changes use POST/PUT/PATCH/DELETE, not GET,
- use synchronizer CSRF tokens for forms/actions,
- use SameSite cookies as a helpful layer, not the only defense,
- verify Origin/Referer for high-risk endpoints when applicable.

21.4.5 Pitfall 5: Trusting client-provided metadata (MIME type, file name, IDs)

Failure mode: Trusting values from the browser:

- file name or extension,
- MIME type,
- IDs or role values in hidden fields,
- “isAdmin” flags,

- prices, discounts, or totals.

Consequences:

- privilege escalation,
- broken authorization (IDOR),
- unsafe file upload storage and path traversal,
- business logic abuse (tampered totals).

Professional defense:

- treat the client as untrusted,
- validate and recompute on the server,
- store authoritative values server-side (services + database),
- use allow-lists for file types and safe storage keys.

21.4.6 Pitfall 6: Insecure direct object references (IDOR)

Failure mode: Using predictable identifiers in URLs or forms and failing to authorize access to that object.
Example:

- `/invoice?id=1234` returns invoice 1234 without verifying ownership.

Professional defense:

- authorize every object access using user/tenant context,
- do not assume “authenticated” implies “authorized”,

- consider opaque IDs where appropriate, but do not rely on obscurity.

```
bool canAccessInvoice(std::int64_t userId, std::int64_t invoiceId) {
  // Must check ownership/tenant membership in the database.
  return invoiceBelongsToUser(userId, invoiceId);
}
```

21.4.7 Pitfall 7: SQL injection and unsafe query construction

Failure mode: Building SQL strings by concatenation, especially with search, sorting, or filtering parameters.

Professional defense:

- always use parameterized queries for values,
- use allow-lists for column names and sort directions (cannot be parameterized safely as values),
- keep SQL isolated in a database layer.

```
// Safe value binding (conceptual; depends on DB library):
auto stmt = db.prepare("SELECT id, name FROM users WHERE email = ?");
stmt.bind(0, email);
```

21.4.8 Pitfall 8: Password storage mistakes

Failure mode:

- storing plaintext or encrypted passwords,
- using fast hashes (MD5/SHA-1/SHA-256) without a password hashing algorithm,
- missing unique salts,

- low-cost bcrypt parameters.

Professional defense:

- use modern password hashing algorithms (Argon2id preferred; bcrypt acceptable),
- store algorithm + parameters + salt + digest in one field,
- rehash on login when parameters are outdated,
- protect login endpoints with rate limiting and MFA where appropriate.

21.4.9 Pitfall 9: Unsafe file upload handling

Failure mode:

- storing uploads under the public web root,
- using user-provided file names as paths,
- serving uploaded files without authorization,
- accepting archives without decompression limits,
- processing complex formats in-process without isolation.

Professional defense:

- store outside web root with generated names,
- enforce size and type allow-lists,
- stream instead of buffering large content,
- scan and/or quarantine when necessary,
- serve downloads through authorization-checked handlers.

21.4.10 Pitfall 10: Cross-thread UI access and race conditions

Failure mode: Accessing session-owned widgets from background threads or shared mutable state without synchronization.

Consequences:

- undefined behavior,
- sporadic crashes,
- corrupted session state,
- security-impacting logic errors.

Professional defense:

- treat `WApplication` and widgets as session-thread-owned,
- post UI updates into the session context using a safe dispatch mechanism,
- keep cross-thread communication message-based and minimal.

21.4.11 Pitfall 11: Logging secrets and sensitive data

Failure mode: Logging:

- session IDs,
- CSRF tokens,
- password reset tokens,
- authorization headers,
- full request bodies with credentials,

- personal data not required for debugging.

Professional defense:

- define a redaction policy,
- log minimal identifiers (hashed/truncated) when correlation is needed,
- separate audit logs from debug logs,
- protect log storage and access.

21.4.12 Pitfall 12: Unbounded resource consumption (DoS by design)

Failure mode:

- unlimited upload sizes,
- unbounded query result sets,
- expensive operations triggered by any user action,
- no timeouts or concurrency limits,
- per-session caches that grow without eviction.

Professional defense:

- enforce size limits everywhere (requests, uploads, exports),
- paginate tables and limit filters,
- rate-limit sensitive endpoints,
- cap concurrent expensive tasks and queue the rest,
- implement bounded caches with TTL/LRU.

21.4.13 Pitfall 13: Unsafe redirects and open redirect vulnerabilities

Failure mode: Redirecting to a URL provided by the client without validation (e.g., `returnUrl` parameter).

Consequences:

- phishing with trusted domain redirects,
- token leakage via redirect chains.

Professional defense:

- allow-list redirect targets (internal paths only),
- reject absolute external URLs,
- normalize and validate before redirect.

21.4.14 Pitfall 14: Incorrect trust boundary behind reverse proxies

Failure mode: Trusting `X-Forwarded-For` and `X-Forwarded-Proto` headers from untrusted sources.

Consequences:

- spoofed IP for rate limit bypass,
- incorrect “HTTPS” detection leading to insecure cookies,
- wrong redirect construction.

Professional defense:

- trust forwarded headers only when the request is known to come from your proxy,
- restrict proxy access by network policy,
- validate and sanitize proxy-provided metadata.

21.4.15 Pitfall 15: “Security by UI” instead of server-side authorization

Failure mode: Hiding buttons or pages in the UI but not enforcing authorization in services. Attackers do not need your UI; they can call endpoints directly.

Professional defense:

- enforce authorization in service methods,
- treat controllers/UI as presentation only,
- verify permissions for every state change and data fetch.

```
Result<Order> OrderService::getOrder(UserId user, OrderId id) {
  if (!authz_.canViewOrder(user, id)) return Result<Order>::forbidden();
  return repo_.loadOrder(id);
}
```

21.4.16 Pitfall 16: Missing secure headers and browser defenses

Failure mode: Neglecting browser security headers even under HTTPS.

Professional baseline:

- Content Security Policy (CSP) where feasible,
- X-Content-Type-Options: nosniff,
- clickjacking protection (frame-ancestors or X-Frame-Options),
- Referrer-Policy.

These headers reduce the blast radius of mistakes (especially XSS).

21.4.17 Pitfall 17: Insecure secrets management

Failure mode:

- embedding secrets in source code,
- committing secrets to repositories,
- using the same secrets across environments,
- long-lived secrets with no rotation plan.

Professional defense:

- store secrets in environment variables or dedicated secret managers,
- rotate secrets periodically and on suspected compromise,
- scope secrets per environment and least privilege.

21.4.18 Pitfall 18: Ignoring security testing and dependency risk

Failure mode: Treating security as a code-only issue and neglecting:

- dependency vulnerabilities (libraries, OS packages),
- misconfigurations at the proxy layer,
- insecure build pipelines,
- lack of automated scanning and patching.

Professional defense:

- keep dependencies updated,

- perform SAST/DAST where appropriate,
- use fuzzing for parsers and file handlers,
- practice incident response drills (backup restore, credential rotation).

21.4.19 High-impact security checklist (production baseline)

- HTTPS everywhere + correct proxy trust model + HSTS rollout plan.
- Secure session cookies + session rotation + server-side invalidation on logout.
- CSRF protection on all state-changing operations.
- XSS-safe rendering defaults + CSP where feasible.
- Authorization enforced in services for every object access (no IDOR).
- Parameterized SQL + allow-listed sort/filter keys.
- Strong password hashing (Argon2id/bcrypt) + rate-limited login + MFA where appropriate.
- Secure upload/download pipeline (limits, storage, authz, scanning).
- Thread-safe architecture (no cross-thread widget access).
- Secrets managed and rotated; logs redacted; monitoring enabled.

21.4.20 Outcome

By systematically avoiding these critical security pitfalls, Wt applications achieve a robust security posture appropriate for production deployments. The objective is not to rely on any single mechanism, but to build layered defenses where failures are contained, anomalies are

detectable, and security controls remain enforceable as the codebase and deployment environment evolve.

Logging and Debugging

22.1 Logging architecture

Logging architecture is the deliberate design of how an application produces, transports, stores, and analyzes operational events. In production-grade Wt systems, logging is not “printf debugging”; it is a core subsystem that supports:

- incident response and forensic analysis,
- security auditing,
- performance investigation and capacity planning,
- correctness validation across distributed components,
- and long-term reliability engineering.

Because Wt is session-based and server-side, logs must also make it possible to understand:

- which session triggered an action,
- which request caused a state change,
- which service call failed,
- how latency is distributed (DB, network, CPU),

- and what the system did under concurrency.

This section describes a professional logging architecture suitable for modern C++ Wt applications.

22.1.1 Design goals

A production logging architecture should satisfy these goals:

- **Correctness**: logs reflect real events and preserve ordering context as much as practical.
- **Usefulness**: logs answer operational questions, not just developer curiosity.
- **Performance**: logging does not become a bottleneck under load.
- **Safety**: logs must not leak secrets, credentials, or sensitive personal data.
- **Consistency**: uniform structure across modules, services, and deployments.
- **Traceability**: events can be correlated across threads, requests, and services.

22.1.2 The three layers of logging

A robust architecture separates logging into three layers:

1) Application logging (domain + technical events)

This is the logging emitted by your code. It includes:

- domain events (e.g., order placed, invoice generated),
- technical events (e.g., DB timeout, validation failure),
- security events (e.g., failed login, access denied).

2) Transport and collection

How logs leave the process:

- stdout/stderr for containerized deployments,
- file appenders with rotation,
- syslog/journald integration,
- log forwarders/agents.

3) Storage, search, and analytics

Where logs are retained and analyzed:

- centralized log systems,
- indexing/search backends,
- retention policies,
- alerting and dashboards.

Even if your book does not name specific vendors, the architecture must assume centralization and structured correlation.

22.1.3 Log taxonomy: what to log

Professional logging divides events into categories with distinct rules:

Operational logs

- startup/shutdown,
- configuration summary (non-secret),
- dependency status (DB reachable, migrations state),
- resource health (pool saturation, memory warnings).

Request and session logs

- request received (method, path, status),
- latency breakdown (total time, DB time),
- session identifier correlation (safe representation only),
- rate-limit and throttling events.

Security audit logs

- authentication success/failure,
- authorization failures,
- password reset initiation and completion,
- suspicious behavior signals (brute-force patterns),
- privilege changes.

Audit logs must be harder to tamper with, often require stricter retention, and must avoid leaking tokens.

Error logs

- exceptions with context,
- failure codes and causal chain,
- dependency errors (timeouts, connection failures),
- invariant violations.

22.1.4 Structured logging: the default professional choice

Plain text logs are human-readable but weak for automation. Structured logging produces logs as key/value fields so you can:

- filter and aggregate reliably,
- correlate events across services,
- compute metrics from logs,
- detect patterns for alerts.

A typical structured event includes:

- timestamp (UTC),
- severity level,
- component/module,
- message (short, stable),
- correlation identifiers (request ID, session ID hash),
- user/tenant identifiers (non-sensitive, minimal),
- event-specific fields (e.g., status code, latency ms).

22.1.5 Correlation and trace context

To debug production issues, you need correlation IDs:

- **Request ID**: unique per inbound request or UI event.
- **Session correlation**: safe session token representation.
- **User ID / tenant ID**: minimal identifiers for authorization context.
- **Span-like phases**: optional, to measure sub-operations.

Because Wt applications can involve multiple UI events per session and multiple internal service calls per event, correlation must be:

- propagated across service boundaries,
- available on background tasks,
- attached to exceptions and error logs.

Safe session correlation

Never log raw session IDs or tokens. Instead log:

- a truncated, hashed representation,
- or a server-generated correlation ID independent of security tokens.

```
#include <string>
#include <cstdint>

static std::string truncate_id(const std::string& s, std::size_t n = 8) {
  if (s.size() <= n) return s;
  return s.substr(0, n);
}
```

22.1.6 Severity levels and policy

Define stable levels and their meaning:

- **TRACE**: extremely detailed, disabled in production.
- **DEBUG**: diagnostic details, typically disabled or sampled.
- **INFO**: normal operations, key state transitions.
- **WARN**: abnormal but recoverable conditions (retries, fallback).
- **ERROR**: operation failed; user-visible impact likely.
- **CRITICAL**: system-wide failure, data integrity risk, immediate alert.

Production policy commonly requires:

- DEBUG/TRACE disabled or sampled,
- structured INFO for main operational events,
- WARN/ERROR for actionable issues,
- CRITICAL triggers paging/alerts.

22.1.7 Performance architecture: avoid logging as a bottleneck

Logging can become a throughput killer if it performs expensive work in hot paths.
Professional patterns:

1) Avoid unnecessary formatting

Only build expensive strings if the log level is enabled.

```
if (logger.is_debug_enabled()) {
  logger.debug("Loaded items=" + std::to_string(count));
}
```

2) Prefer asynchronous logging for high volume

Async logging decouples request latency from log I/O:

- producer threads enqueue events,
- a dedicated consumer thread flushes to sink (stdout/file/syslog).

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <thread>

struct LogEvent {
  std::string line;
};

class AsyncLogger {
public:
  AsyncLogger() : worker_([this]{ run(); }) {}

  ~AsyncLogger() {
    {
      std::lock_guard<std::mutex> lk(m_);
      stop_ = true;
    }
```

```
    cv_.notify_one();
    worker_.join();
  }

  void info(std::string msg) {
    std::lock_guard<std::mutex> lk(m_);
    q_.push(LogEvent{std::move(msg)});
    cv_.notify_one();
  }

private:
  void run() {
    std::unique_lock<std::mutex> lk(m_);
    while (!stop_ || !q_.empty()) {
      cv_.wait(lk, [&]{ return stop_ || !q_.empty(); });
      while (!q_.empty()) {
        auto e = std::move(q_.front());
        q_.pop();
        lk.unlock();
        // Write to sink (stdout/file). Keep it fast.
        // sink_write(e.line);
        lk.lock();
      }
    }
  }

  std::mutex m_;
  std::condition_variable cv_;
  std::queue<LogEvent> q_;
  bool stop_{false};
  std::thread worker_;
};
```

3) Bounded queues and backpressure

Async log queues must be bounded:

- if the sink is slow (disk pressure), unlimited queues cause memory blow-up,
- define behavior when full: drop DEBUG, sample, or block briefly.

4) Separate audit logs

Audit logs should not be dropped silently. Treat them with stricter guarantees:

- separate sink,
- stronger retention,
- stable format,
- minimal performance overhead but high reliability.

22.1.8 Security architecture: redaction and minimization

Logging is a major data leak vector. Adopt a strict rule set:

- never log passwords, raw tokens, session IDs, CSRF tokens,
- never log authorization headers,
- redact sensitive fields in request bodies (email may be acceptable, but minimize),
- avoid logging full SQL queries with user data embedded,
- avoid storing PII unless necessary for support and compliance.

Implement a redaction layer:

- a sanitizer function for key/value events,
- a structured logging API that classifies fields as sensitive or safe.

22.1.9 Wt-specific logging integration points

In Wt systems, useful logging points include:

- session creation/destruction,
- login/logout transitions,
- major UI actions (submit order, upload file),
- model validation failures (with safe field names, not values),
- service layer failures (DB timeout, external API failure),
- server push updates (if used) and their rate.

Avoid logging on every UI micro-event (e.g., every keystroke) unless sampled, because it creates volume and can degrade performance.

22.1.10 Debugging architecture: logs + metrics + traces

Professional debugging does not rely on logs alone. Architecture combines:

- **Logs**: discrete events with context.
- **Metrics**: counters and distributions (latency, pool usage).
- **Traces**: request flows across components.

Even if you implement only logs initially, design log fields so they can be correlated with metrics/traces later (request ID, component, latency fields).

22.1.11 Rotation, retention, and operational policy

If logs are stored locally:

- rotate by size/time,
- compress old logs,
- enforce retention limits,
- monitor disk usage.

If logs are centralized:

- define retention policy per log class (audit vs debug),
- control access (least privilege),
- ensure tamper resistance for audit trails.

22.1.12 A practical logging interface for a Wt application

A small, structured API keeps log usage consistent:

```
#include <string>
#include <map>

enum class Level { Info, Warn, Error };

struct LogRecord {
  Level level{Level::Info};
  std::string component;
  std::string message;
  std::map<std::string, std::string> fields; // structured payload
};
```

```
class ILogger {
public:
  virtual ~ILogger() = default;
  virtual void write(const LogRecord& r) = 0;
};

static std::string level_to_string(Level l) {
  switch (l) {
    case Level::Info:  return "INFO";
    case Level::Warn:  return "WARN";
    case Level::Error: return "ERROR";
  }
  return "INFO";
}
```

With such an API, you can enforce:

- redaction rules at a single point,
- consistent correlation fields,
- stable formatting for sinks.

22.1.13 Checklist: logging architecture readiness

- Structured logs with stable fields (time, level, component, request ID).
- Correlation strategy (request ID + safe session correlation).
- Clear severity policy and production filtering/sampling plan.
- Async logging or optimized sinks to avoid request latency spikes.
- Bounded queues and defined behavior under pressure.

- Separate audit logging channel with stronger retention and access control.
- Redaction policy implemented and tested (no secrets in logs).
- Rotation/retention policy defined and monitored.
- Logs aligned with debugging needs (paired with metrics/traces where possible).

22.1.14 Outcome

A well-designed logging architecture makes Wt applications operable at scale. It provides high-signal diagnostics without harming performance, protects sensitive information, supports auditing, and enables rapid debugging under concurrency. The result is a production system that can be trusted not only to run correctly, but to be understood when something goes wrong.

22.2 Error tracing

Error tracing is the systematic collection, propagation, and correlation of error context across layers of a Wt application. In production systems, simple logs of exceptions are not enough to understand root cause. Error tracing ties errors to the conditions that caused them: the session context, request metadata, service call chains, and external dependencies. It also enables post-mortem analysis, operational alerts, and corrective automation.
This section outlines a professional error tracing architecture for modern C++ Wt applications, including structured propagation, context enrichment, correlation identifiers, stack capture, and integration with centralized observability systems.

22.2.1 Goals of error tracing

Error tracing supports:

- pinpointing where and why errors occur,

- linking application and dependency errors across services,
- reducing mean time to resolution (MTTR),
- triggering automated alerts on patterns,
- providing contextual information that avoids log hunting.

In a Wt application, error tracing must be designed for:

- session-affine event flows,
- server-side callbacks and asynchronous work,
- service/business logic layers,
- and external subsystem interactions (DB, storage, APIs).

22.2.2 Structured error context

A minimal structured error event should include:

- timestamp (UTC),
- severity level,
- short error message,
- error type or code (business vs system),
- stack trace or location identifier,
- request ID,
- session correlation ID (safe representation),

- user/tenant context (when available),
- component/service name,
- additional fields (dependency, latency, parameters).

This structure enables automated querying, filtering, and graphing.

22.2.3 Error categories

Not all errors are equal. Classify errors before tracing:

- **Validation errors**: incorrect user input,
- **Business rule violations**: expected domain failures,
- **System errors**: infrastructure issues (DB timeout),
- **Unexpected exceptions**: logic bugs or unforeseen states,
- **External API failures**: dependency communication issues.

Categorization influences tracing, alert thresholds, and operational responses.

22.2.4 Capturing context at the throw site

Where an error occurs (within business logic, controller, or repository), capture rich context:

```
struct ErrorContext {
  std::string message;
  std::string component;
  std::string location; // file:line or function
  std::string errorCode;
  std::map<std::string, std::string> fields;
};
```

When throwing or tagging an error, include context values that help diagnose without exposing sensitive data.

22.2.5 Propagation and enrichment

As an error propagates up the call stack, context should be enriched, not replaced:

- repository adds query details,
- service adds business rule identifiers,
- controller adds request/session identifiers.

A chained context model avoids losing information:

```
struct TracedError {
  ErrorContext base;
  std::vector<ErrorContext> chain;
};
```

Each layer pushes context into the chain.

22.2.6 Stack and location identification

Capturing a stack trace is valuable for unexpected exceptions. In C++:

- use platform APIs or libraries that can capture symbolized stacks,
- store location identifiers (function/file/line),
- avoid capturing stack on expected failures (costly).

```
// Pseudocode conceptual example
auto stack = capture_stack_trace();
record["stackTrace"] = stack.to_string();
```

22.2.7 Correlation identifiers

An error event is most useful when correlated with:

- a unique request ID,
- a session correlation ID,
- a user or tenant identifier (safe and minimal),
- and possibly a trace ID if distributed tracing is used.

Propagation of correlation IDs across layers and asynchronous boundaries is essential.

```
struct Correlation {
  std::string requestId;
  std::string sessionIdHash;
  std::string userId;
};
```

22.2.8 Integration with structured logs

Error tracing emits structured events that can be indexed and aggregated. A typical log event object might be:

```
struct ErrorLogEvent {
  std::string timestamp;
  std::string severity;
  Correlation correlation;
  TracedError error;
};
```

A logging backend (file, syslog, or centralized system) ingests this for search and analysis.

22.2.9 Error tagging and marking expected vs unexpected

Professional error tracing distinguishes:

- **Expected errors**: validation fail, business conditions (non-fatal),
- **Unexpected errors**: exceptions that indicate correctness or integration faults.

Expected errors typically log at INFO or WARN with reduced stack detail. Unexpected errors log at ERROR or CRITICAL with full context.

22.2.10 Error catch and report boundaries

Common catch points include:

- at the entry to controllers to wrap all uncaught exceptions,
- at service boundaries to tag dependency context,
- at asynchronous dispatch boundaries to ensure errors do not silently drop.

A typical pattern:

```
try {
  auto result = service.doWork(req);
} catch (const std::exception& ex) {
  TracedError te = enrich_with_context(ex, "ServiceLayer", correlation);
  logger.write({current_timestamp(), "ERROR", correlation, te});
  throw; // propagate or map to safe UI error
}
```

22.2.11 Asynchronous and background work

Background jobs, worker threads, and scheduled tasks must also propagate correlation:

- pass correlation IDs into the work context,
- attach them to logs and tracing events,
- ensure errors in background jobs produce operational alerts.

```
void processBackground(const Correlation& corr) {
  try {
    doTask();
  } catch (...) {
    logErrorWithCorrelation(corr, "Background job failed");
  }
}
```

22.2.12 Linking traces across dependent services

In multi-service architectures:

- propagate trace IDs in request headers,
- include them in logs on both caller and callee,
- correlate events in aggregated backends.

Even if your system is monolithic, linking errors from DB or storage adapters with application context is crucial.

22.2.13 Semantic tagging of error fields

Include fields that capture non-sensitive but actionable data:

- HTTP status code,
- DB error codes,
- user role and tenant (abstracted),
- feature flags or configuration variants,
- latency measurements.

22.2.14 Error dashboards and alerting

Operationally, tracing is useful only if integrated with:

- alert thresholds (rate of ERROR/CRITICAL),
- dashboards summarizing error types by endpoint/service,
- time windows for spikes and trends,
- filters for noise reduction and grouping by root cause.

22.2.15 Privacy and compliance in error traces

Protect sensitive data:

- never include raw passwords, tokens, or secrets,
- avoid logging full request bodies with PII,
- apply field redaction rules,
- mask identifiers appropriately.

22.2.16 Debuggable failures without verbose logging

Logging everything at full detail is costly. Use sampling and guard conditions:

- sample repetitive, low-impact errors,
- escalate on patterns (e.g., repeating failures per session),
- flag unusual error mixes for deeper capture.

22.2.17 Testing error tracing

Tests should verify:

- error context is captured and propagated,
- correlation IDs flow across service boundaries,
- stack traces are present for unexpected errors,
- sensitive fields are never logged.

Example:

```
void test_error_context_propagation() {
  Correlation corr{"req1","sessA","userX"};
  try {
    throw std::runtime_error("fail");
  } catch (...) {
    auto traced = enrich_with_context(std::current_exception(), "Test", corr);
    assert(traced.chain.size() > 0);
  }
}
```

22.2.18 Checklist: error tracing readiness

- Structured error events with context fields.
- Correlation identifiers propagated across layers and async work.
- Distinction between expected and unexpected errors.
- Stack/location information where appropriate.
- Integration with centralized logging/alerting systems.
- Redaction and privacy controls.
- Tests verifying propagation and safety.

22.2.19 Outcome

A professional error tracing architecture turns error logs into actionable insights. It bridges the gap between low-level exceptions and high-level system behavior, enabling teams to rapidly diagnose problems, understand cross-layer interactions, and maintain resilient Wt applications in production environments.

22.3 Production diagnostics

Production diagnostics is the engineering discipline of observing, measuring, and debugging a live system *without* disrupting its availability, correctness, or security posture. In a production-grade Wt application, diagnostics must work under real load, across many concurrent sessions, and in the presence of failures that may be intermittent or non-reproducible in development.
A professional diagnostics approach has four objectives:

- **Detect**: know when something is wrong (availability, latency, correctness, security).
- **Localize**: determine where the problem originates (UI event, controller, service, DB, storage, network).
- **Explain**: capture enough context to understand why it happened (correlation IDs, state, timings).
- **Mitigate**: reduce impact quickly (feature flags, throttling, graceful degradation) while preparing a fix.

22.3.1 Diagnostics pillars: logs, metrics, traces, and profiles

Production diagnostics is strongest when built on four pillars:

1) Logs

- discrete events and errors with context,
- useful for post-mortem and forensic reasoning,
- must be structured and redacted.

2) Metrics

- numeric signals over time (counters, gauges, histograms),
- ideal for alerts and trend detection,
- extremely low overhead when designed correctly.

3) Traces

- end-to-end request/event flow across layers,
- best for latency decomposition and dependency diagnosis,
- requires correlation context propagation.

4) Profiles

- CPU and memory profiling to pinpoint hot paths and leaks,
- generally used during incident response windows or controlled sampling,
- must be safe and bounded to avoid destabilizing the service.

22.3.2 Production-safe diagnostic design principles

Principle 1: Never leak secrets

Diagnostics must not expose:

- passwords,
- session identifiers,
- CSRF tokens,
- authorization headers,
- private keys,
- full request bodies containing sensitive data.

Apply redaction by default, not as a best-effort habit.

Principle 2: Bounded overhead

Diagnostics must be engineered with explicit limits:

- sampling (e.g., trace 1% of events),
- rate limits on expensive logging,
- bounded queues for async logging,
- timeouts for diagnostic endpoints.

Principle 3: Deterministic correlation

Every diagnostic artifact must support correlation:

- request/event ID,
- session correlation ID (safe form, not raw),
- user/tenant identifier (minimal),
- component name and version/build ID.

22.3.3 Health endpoints and service readiness

Production systems require two distinct health concepts:

Liveness

Answers: “Is the process running and able to respond?”

- must be fast,
- must not depend on external services,
- should not perform heavy checks.

Readiness

Answers: “Is the service ready to receive real traffic?”

- may check DB connectivity, migration state, critical dependencies,
- must be bounded by strict timeouts,
- should return degraded status when dependencies are unhealthy.

```
#include <chrono>
#include <string>
#include <optional>

struct HealthStatus {
  bool ok{true};
  std::string detail;
};

class Diagnostics {
public:
  HealthStatus liveness() const {
    return HealthStatus{true, "ok"};
  }

  HealthStatus readiness() const {
    // Example: bounded dependency check
   const auto deadline = std::chrono::steady_clock::now() + std::chrono::milliseconds(200);
    if (!db_ping(deadline)) {
      return HealthStatus{false, "db_unreachable"};
    }
    return HealthStatus{true, "ready"};
  }
```

```
private:
  bool db_ping(std::chrono::steady_clock::time_point deadline) const {
    // Implement a non-blocking or timeout-respecting ping.
    // Return false if ping cannot complete before deadline.
    (void)deadline;
    return true;
  }
};
```

In Wt, health endpoints are commonly served by a lightweight HTTP handler at the reverse proxy or connector layer, not by expensive session-based UI logic.

22.3.4 Key production metrics for Wt applications

A Wt application is session-based; therefore, you must monitor not only request rate, but also session load and server resources.

Latency metrics

Track latency distributions (not just averages):

- total request/UI-event time,
- DB time,
- external API time,
- queue time waiting for worker threads,
- rendering/diff generation time (if measurable).

Session metrics

- active sessions,
- idle sessions,
- session churn rate (new sessions/minute),
- per-session memory estimates (sampled),
- session timeouts and abnormal disconnects.

Resource metrics

- CPU utilization,
- RSS memory, heap usage (if available), fragmentation signals,
- file descriptors / sockets,
- thread count and thread pool queue depth,
- DB pool utilization and wait time.

Error metrics

- error rate by endpoint/action,
- error rate by error code/category,
- authentication failures and rate-limit triggers,
- timeouts and retries.

```cpp
#include <atomic>
#include <chrono>

struct Metrics {
  std::atomic<std::uint64_t> requests_total{0};
  std::atomic<std::uint64_t> errors_total{0};
  std::atomic<std::uint64_t> sessions_active{0};

  // Lightweight timing helper for histograms (conceptual)
  static std::uint64_t now_ns() {
    return static_cast<std::uint64_t>(
      std::chrono::duration_cast<std::chrono::nanoseconds>(
        std::chrono::steady_clock::now().time_since_epoch()
      ).count()
    );
  }
};
```

22.3.5 Structured error reporting and incident-grade context

Production diagnostics is strongest when errors are reported with:

- a stable error code,
- a short message intended for operators,
- correlation IDs,
- non-sensitive context fields (dependency name, timeout ms, query class),
- optional stack trace for unexpected failures.

A professional system distinguishes:

- expected failures (validation, business rules): usually INFO/WARN, no stack,

- unexpected failures (bugs, invariants, crashes): ERROR/CRITICAL, include stack and build fingerprint.

```
#include <map>
#include <string>

struct Correlation {
  std::string request_id;
  std::string session_tag; // safe; not raw session ID
  std::string user_id;     // minimal
};

struct ErrorEvent {
  std::string code;        // stable: e.g. "DB_TIMEOUT"
  std::string message;     // operator-focused
  std::string component;   // "OrderService"
  Correlation corr;
  std::map<std::string, std::string> fields;
};
```

22.3.6 Runtime debug switches (feature flags and diagnostic toggles)

A production system must allow controlled diagnostic escalation without redeploying:

- increase log level for a single component,
- enable tracing for a subset of users/sessions,
- enable extra validation for a single endpoint,
- temporarily disable a problematic feature (kill switch),
- throttle heavy operations.

Rules for safe toggles:

- toggles must be authenticated and authorized,
- toggles must be auditable (who changed what and when),
- toggles must be time-bounded (automatic expiry),
- toggles must not expose secrets.

```
#include <atomic>

struct DiagnosticToggles {
  std::atomic<bool> enable_extra_tracing{false};
  std::atomic<int>  component_log_level{2}; // 0..5, for example
  std::atomic<bool> kill_switch_exports{false};
};
```

22.3.7 Crash diagnostics: core dumps and post-mortem analysis

For native C++ servers, crashes can be diagnosed via:

- core dumps (or platform equivalents),
- symbolized stack traces (with matching binaries),
- build IDs and symbol packages,
- allocator diagnostics (double free, heap corruption).

Production posture:

- ensure builds can be symbolized (separate debug symbols),
- record build fingerprint in logs,

- configure core dump handling securely (cores contain secrets in memory),
- prefer rapid restart with preservation of diagnostic artifacts.

22.3.8 Memory diagnostics in production

Memory issues commonly appear as:

- slow leaks (RSS increases gradually),
- fragmentation (RSS grows even when allocations churn),
- per-session retention (inactive sessions keep large objects),
- runaway caches (no eviction),
- accidental buffering (uploads/exports in memory).

Production-safe techniques:

- periodic sampling of heap metrics (if supported),
- high-water mark logging at coarse intervals,
- per-session footprint sampling on a small fraction of sessions,
- strict limits on caches, uploads, and exports.

```
#include <cstddef>
#include <string>

struct MemorySnapshot {
  std::size_t rss_bytes{0};
  std::size_t heap_estimate_bytes{0}; // if available
  std::size_t sessions_active{0};
  std::string note;
};
```

22.3.9 Latency diagnostics: finding where time goes

To diagnose latency, you need phase timing:

- controller entry to response end,
- service call durations,
- DB query durations,
- serialization/diff durations,
- queue wait times.

Avoid logging every timing for every request. Instead:

- record histograms in metrics,
- emit detailed timing logs only on slow requests (threshold-based),
- sample traces for representative flows.

```
#include <chrono>
#include <cstdint>

class ScopedTimer {
public:
  explicit ScopedTimer(std::uint64_t& out_ns)
    : out_(out_ns), start_(std::chrono::steady_clock::now()) {}

  ~ScopedTimer() {
    const auto end = std::chrono::steady_clock::now();
    out_ = static_cast<std::uint64_t>(
      std::chrono::duration_cast<std::chrono::nanoseconds>(end - start_).count()
    );
```

```
  }

private:
  std::uint64_t& out_;
  std::chrono::steady_clock::time_point start_;
};
```

22.3.10 Threading and concurrency diagnostics

Concurrency problems in Wt often manifest as:

- long queueing delays (thread pool saturation),
- deadlocks or lock contention,
- incorrect cross-thread access to session-owned UI state,
- background tasks starving interactive events.

Production diagnostics should include:

- thread pool queue depth gauge,
- request queue time histogram,
- lock contention sampling (where practical),
- watchdog for stalled requests (timeouts with correlation IDs).

22.3.11 Dependency diagnostics: database, storage, and external APIs

Most production incidents are dependency-driven. For each dependency, track:

- success/failure rate,

- latency distributions,
- timeout count,
- retry count,
- circuit breaker open/half-open state (if used),
- pool utilization (DB connections, HTTP clients).

Engineering patterns:

- explicit timeouts everywhere (no infinite waits),
- bounded retries with jitter,
- graceful degradation paths,
- clear error codes and operator messages.

22.3.12 Safe production debugging workflow

A professional workflow reduces risk during incidents:

1. Confirm impact using metrics (error rate, latency, saturation).
2. Identify the scope (which endpoints, which sessions, which dependency).
3. Enable targeted diagnostics:
    - increase log level for one component,
    - enable tracing for one request ID or a small sample,
    - enable slow-request detail logging above threshold.

4. Apply mitigation:
    - throttle heavy operations,
    - disable a non-critical feature,
    - reduce export size or pagination limits temporarily.
5. Capture artifacts (logs, traces, core dumps if crash).
6. Roll forward with a fix; roll back only if necessary and safe.
7. Post-mortem: improve signals, add tests, add guardrails.

22.3.13 Production diagnostics checklist

- Liveness and readiness endpoints with strict timeouts.
- Structured logs with correlation IDs and redaction.
- Metrics for latency (p50/p95/p99), errors, sessions, pools, resources.
- Trace sampling with request/session correlation propagation.
- Threshold-based “slow request” diagnostic events.
- Safe runtime toggles (feature flags, component log level) with auditing and expiry.
- Crash strategy: symbolized stacks, secure core handling, build fingerprint.
- Memory and cache bounds; upload/export streaming; eviction policies.
- Dependency instrumentation (DB/API/storage) with timeouts and retry metrics.
- Operational runbooks describing how to escalate diagnostics safely.

22.3.14 Outcome

When production diagnostics is engineered as a first-class subsystem, a Wt application becomes observable and operable under real-world constraints. Incidents are detected early, localized quickly, explained with high-signal context, and mitigated safely. The result is not only easier debugging, but higher uptime, stronger security, and a system that remains stable as traffic and complexity grow.

Part X

Deployment on Linux

Running Wt as a Server Application

23.1 Standalone server mode

Standalone server mode means your Wt application binary *is itself* the HTTP/WebSocket server. Instead of running behind a separate application connector (such as FastCGI) that is hosted inside a general-purpose web server, the program links against Wt's built-in HTTP(S) connector and listens on a TCP port. In this mode, the binary can serve:

- the Wt dynamic application (sessions, widgets, events, server push),
- WebSocket connections (when enabled),
- and a limited set of static resources (depending on configuration).

On Linux, standalone mode is a common deployment choice when you want a simple and predictable runtime model: one process (or a small number of processes), explicit ports, explicit thread usage, and a clean integration with `systemd`. In production, it is still typical to place a reverse proxy in front (for TLS termination, buffering, security headers, and routing), but the Wt process remains the primary application server.

23.1.1 When standalone mode is a good fit

Standalone mode is a strong choice when:

- You want a single self-contained executable that runs as a service.
- You want direct control over the Wt server configuration, threading model, and lifecycle.
- You rely on WebSockets and want an uncomplicated server-side setup.
- You prefer reverse proxy + application server architecture (common on Linux).
- You want to avoid FastCGI operational complexity and connector-specific issues.

23.1.2 Trade-offs versus FastCGI deployment

Standalone mode is not automatically superior; it is a different operating model.

Standalone mode advantages

- Fewer moving parts: one binary that listens and serves.
- Clean WebSocket support (single hop) and predictable session behavior.
- Easier local reproduction of production behavior (same server stack).
- Straightforward `systemd` integration and service supervision.

Standalone mode trade-offs

- A general-purpose web server still adds value for TLS, static asset caching, WAF rules, and routing.
- You must plan scaling explicitly (multiple instances, reverse proxy load balancing, and session affinity if needed).
- Your process now owns both application logic and server connection handling; misconfiguration can affect availability.

23.1.3 Core concept: the built-in HTTP(S) connector

In Wt, linking the application against the built-in HTTP(S) connector produces a binary that can act as a stand-alone HTTP/WebSockets server. Operationally, that binary:

- reads server configuration from command-line options, a configuration file, or both,
- constructs a `Wt::WServer` instance,
- registers your application entry point (factory),
- and runs until shutdown is requested.

23.1.4 The typical entry point: WRun and WServer

For many applications, `Wt::WRun()` is the standard entry point. It is a convenience helper that creates and configures a `Wt::WServer` and runs the event loop until shutdown.

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WRun.h>

class MyApp final : public Wt::WApplication {
public:
  explicit MyApp(const Wt::WEnvironment& env)
    : Wt::WApplication(env)
  {
    setTitle("Standalone Wt Server");
    root()->addWidget(std::make_unique<Wt::WText>("Hello from Wt standalone mode."));
  }
};

static std::unique_ptr<Wt::WApplication> createApplication(const Wt::WEnvironment& env) {
```

```
  return std::make_unique<MyApp>(env);
}

int main(int argc, char** argv) {
  return Wt::WRun(argc, argv, &createApplication);
}
```

For advanced control (multi-app mounting, explicit error handling, custom initialization), you can build and manage `Wt::WServer` explicitly:

```
#include <Wt/WServer.h>
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>

static std::unique_ptr<Wt::WApplication> createApplication(const Wt::WEnvironment& env);

int main(int argc, char** argv) {
  try {
    Wt::WServer server;

    // Configure using command-line and/or configuration file:
    // The configuration defines HTTP(S) connector settings and Wt runtime behavior.
    server.setServerConfiguration(argc, argv);

    // Register the application at "/" (or another path).
    server.addEntryPoint(Wt::EntryPointType::Application, &createApplication);

    if (server.start()) {
      // Block until shutdown is requested, then stop cleanly.
      Wt::WServer::waitForShutdown();
      server.stop();
    }
  } catch (const std::exception& e) {
```

```
    // Log to stderr; in production, prefer structured logging.
    // std::cerr << "Fatal: " << e.what() << "\n";
    return 1;
  }
  return 0;
}
```

23.1.5 Configuration sources and precedence

A professional standalone deployment uses a configuration file (commonly `wt_config.xml`) and a minimal set of command-line overrides. The server can be configured by:

- command-line arguments (useful for ports, document root, runtime paths),
- a configuration file (recommended for stable production configuration),
- or both (with a clear precedence policy).

The configuration file is also the correct place to define per-application settings (selected by application path selectors) and shared defaults.

23.1.6 Built-in server responsibilities and static resources

Standalone mode can serve static resources, but production architecture usually delegates heavy static traffic to a reverse proxy. Your goals should be:

- keep the Wt process focused on dynamic sessions and business logic,
- serve only what is necessary from Wt directly (or only during development),
- let the reverse proxy handle caching, compression, and large static content.

A common pitfall is accidentally routing static resource requests through the Wt application entry point, which may create extra sessions or waste resources. The professional approach is:

- define clear static paths handled by the server/proxy,
- ensure Wt session creation is reserved for real application entry points,
- validate behavior under load (static storms should not exhaust sessions).

23.1.7 Threading and concurrency model in standalone mode

In standalone mode, Wt handles:

- accepting connections,
- dispatching events,
- session management,
- and (optionally) WebSocket traffic.

Key operational points:

- The **session model** is stateful: sessions may persist server-side, holding UI state.
- Wt executes UI callbacks and event handling within its threading model; your code must respect it.
- Long blocking operations inside UI event handlers reduce throughput and increase latency.

Professional guidance:

- Keep UI callbacks small and fast.
- Offload heavy work to background workers (with safe handoff back to the session context).

- Use connection pools for DB and external APIs to avoid per-request connection overhead.
- Monitor thread pool saturation and queue delays as first-class metrics.

23.1.8 Ports, binding, and Linux networking posture

Standalone mode means binding to a port (e.g., 8080). Production posture:

- Bind the Wt server to `127.0.0.1` or an internal interface when using a reverse proxy.
- Expose only the reverse proxy to the internet (usually port 443).
- Use firewall rules to ensure the Wt port is not reachable externally.

This isolates the application server from direct internet traffic and centralizes security controls at the edge.

23.1.9 Reverse proxy fronting (recommended production layout)

Even when running standalone, the standard production model is:

- Reverse proxy (TLS termination, HSTS, header policy, buffering)
- $\rightarrow$ Wt standalone server (HTTP on loopback/private network)

Benefits:

- stable TLS operations and automated certificate renewal,
- request buffering and protection against slow clients,
- compression and caching,
- consistent security headers,
- easier multi-instance load balancing (when scaling out).

23.1.10 systemd service integration

Running standalone mode professionally on Linux means running it as a supervised service.

```
[Unit]
Description=My Wt Standalone Server
After=network.target

[Service]
Type=simple
User=wtapp
Group=wtapp
WorkingDirectory=/opt/myapp
ExecStart=/opt/myapp/myapp --config /etc/myapp/wt_config.xml
Restart=on-failure
RestartSec=2

# Hardening (adjust to your needs)
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=strict
ProtectHome=true
ReadWritePaths=/var/lib/myapp /var/log/myapp
AmbientCapabilities=
CapabilityBoundingSet=

[Install]
WantedBy=multi-user.target
```

Operational recommendations:

- Run as a dedicated unprivileged user.
- Keep config in `/etc`, state in `/var/lib`, logs in `/var/log`.

- Use `Restart=on-failure` and monitor restart loops (they signal real faults).

23.1.11 Graceful shutdown behavior

A professional server must shut down gracefully:

- stop accepting new work,
- allow active sessions to complete within a policy window,
- flush logs,
- close resources cleanly (DB pools, file handles).

In standalone mode, the Wt server run loop typically returns only after shutdown is signaled and active sessions are properly shut down. This aligns well with `systemd` stop semantics.

23.1.12 Deployment checklist for standalone mode

- Standalone build links against the built-in HTTP(S) connector.
- Configuration defined in `wt_config.xml` with minimal CLI overrides.
- Reverse proxy terminates TLS; Wt binds to loopback/private interface.
- Session and thread settings are validated under peak load.
- Static assets are served by the proxy (or carefully partitioned paths in Wt).
- Structured logging and correlation IDs enabled; secrets redacted.
- systemd unit deployed with hardening and correct file permissions.
- Health checks (liveness/readiness) exist and are fast.
- Monitoring includes sessions, latency percentiles, pool saturation, and error rates.

23.1.13 Outcome

Standalone server mode makes Wt deployment clean and deterministic: your application is the server, with explicit configuration, explicit lifecycle, and a natural fit for Linux service management. With a reverse proxy in front and disciplined session/thread design, standalone mode supports production-grade reliability, security, and performance while keeping the operational model simple.

23.2 Ports and networking basics

Understanding ports and networking fundamentals is essential when deploying a Wt application in standalone server mode on Linux. Networking is the foundation that connects clients (browsers, APIs) to your server, and incorrect configuration at this layer can lead to connectivity failures, security vulnerabilities, or unexpected behavior under load. This section explains the core concepts of TCP/UDP ports, binding interfaces, firewalls, reverse proxying, port ranges, and practical deployment patterns for production Wt applications.

23.2.1 TCP and UDP basics

At the transport layer of the internet protocol stack, two primary protocols are used:

- **TCP (Transmission Control Protocol)**: connection-oriented, reliable delivery, ordered byte streams. Wts HTTP and WebSocket traffic runs over TCP.
- **UDP (User Datagram Protocol)**: connectionless, no delivery guarantees, lower overhead. UDP is not used by Wt for HTTP; it is relevant for DNS, monitoring heartbeats, or custom protocols if integrated.

For web traffic, TCP ports are the focus because:

- HTTP/1.1, HTTP/2, and WebSocket all ride on TCP sockets.

- Reliable delivery and ordered semantics matter for session consistency.
- Reverse proxies and load balancers expect a TCP transport.

23.2.2 What is a port?

A **port** is a 16-bit number (065535) that identifies an endpoint within a host IP address. Every TCP connection is defined by a tuple:

$$(\text{source IP}, \text{source port}, \text{destination IP}, \text{destination port})$$

When your Wt server listens on a port (e.g., 8080), it is saying:

- Accept TCP connections destined for this IP and this port.

Common well-known ports include:

- 80: HTTP (cleartext web traffic),
- 443: HTTPS (encrypted web traffic),
- 8080: common application server port for dev/testing,
- 30005000: commonly used for application servers in microservices.

Ports below 1024 are **privileged** on Unix/Linux and generally require root privileges to bind.

23.2.3 Binding addresses and interfaces

When a server binds a port, it specifies:

- a **port number** (e.g., 8080),
- an **interface or IP address** to bind (e.g., 0.0.0.0, 127.0.0.1, specific NIC IP).

0.0.0.0 Bind all IPv4 interfaces. The server will accept connections on any network interface assigned to the host.

127.0.0.1 (loopback) Bind only to the local host. Useful when running behind a reverse proxy. The application server does not accept direct public requests.

Specific interface (e.g., 10.0.1.5) Bind only to that NIC. Useful in multi-NIC environments to isolate service exposure.
Practical binding rule:

- In production with a reverse proxy (e.g., Nginx), bind Wt to `127.0.0.1` or a private IP and let the proxy listen on 80/443.
- Do not bind directly to public interfaces unless you manage TLS and security headers in the application.

23.2.4 Reverse proxying and port redirection

In production, a common pattern is:

- **Reverse proxy** (Nginx/Apache/HAProxy) listens on ports 80/443,
- **Wt standalone** listens on an internal port (e.g., 8080),
- The proxy forwards requests to the Wt server.

Example flow:

```
Client (HTTPS) -> Proxy:443 -> Internal (HTTP) to Wt:8080
```

Benefits:

- TLS is terminated at the proxy (centralized certificate management),

- static assets can be cached/served by the proxy,
- the application server is isolated from internet traffic,
- port 80/443 require no elevated privileges inside the app.

23.2.5 Firewalls and access control

On Linux, packets are filtered using systems like `iptables`, `nftables`, or host-level firewalls. For secure deployment:

- Allow only designated ports (e.g., 443 for HTTPS) from untrusted networks.
- Block direct access to internal application ports (e.g., 8080) from public networks.
- Allow monitoring and health check ports only from trusted sources (load balancers).

A minimal firewall rule concept:

```
// Allow HTTPS
iptables -A INPUT -p tcp --dport 443 -j ACCEPT

// Block direct access to internal app port
iptables -A INPUT -p tcp --dport 8080 -j DROP

// Allow localhost to internal port
iptables -A INPUT -p tcp -s 127.0.0.1 --dport 8080 -j ACCEPT
```

Adjust rules based on network architecture.

23.2.6 Port ranges and ephemeral ports

When your Wt server makes outbound connections (e.g., to a database, REST API), the kernel selects an **ephemeral port** from a defined range (e.g., 3276860999). This is separate from listening ports.

For high outbound fan-out:

- ensure the ephemeral port range is large enough,
- avoid port exhaustion by reusing connections (connection pools),
- monitor FIN/WAIT states to avoid socket buildup.

23.2.7 Hostnames, DNS, and virtual hosts

Clients connect using hostnames (e.g., `api.example.com`). On the server side:

- DNS resolves the hostname to an IP address,
- the reverse proxy uses Host headers to route to the appropriate application instance (virtual hosting),
- Wt can also inspect Host headers for multi-tenant routing if needed.

Ensure DNS TTLs are appropriate for your deployment pattern (e.g., short TTLs for blue/green switches).

23.2.8 Health check endpoints and monitoring ports

Production deployments often expose:

- **Health check endpoints** (liveness/readiness) on the same port, but often under specific paths (e.g., `/healthz`),
- **Metrics endpoints** (for Prometheus or other collectors),
- **Debug or profiling endpoints** (secured and rarely exposed publicly).

Configure reverse proxy rules to allow health checks from trusted monitors while keeping debug endpoints restricted.

23.2.9 TLS and port assignment

By convention:

- port 80 is reserved for HTTP,
- port 443 is reserved for HTTPS.

If the Wt server handles TLS directly (rare in production), it must:

- bind to port 443 (or another designated HTTPS port),
- manage certificates and private keys,
- configure ciphers and protocols appropriately.

In standalone mode behind a proxy, Wt usually only sees HTTP traffic on an internal port.

23.2.10 IPv4 and IPv6 considerations

Modern deployments should support both IPv4 and IPv6 where available. When binding:

- explicitly bind to IPv4 and IPv6 addresses (e.g., `0.0.0.0` and `::`),
- test connectivity using both families,
- ensure firewall rules cover both IPv4 and IPv6.

Example binding string in a configuration:

```
--http-address 0.0.0.0 --http-address :: --http-port 8080
```

23.2.11 Session affinity and load balancing

When scaling out (multiple Wt instances), clients may connect through a load balancer. For session-stateful applications:

- enable **session affinity** (sticky sessions) at the proxy/load balancer,
- or externalize session state into a shared cache/database for true stateless scaling,
- ensure affinity cookies or hashes are secure and not easily spoofed.

Affinity is implemented at networking layers (reverse proxy, load balancer) typically via:

- IP hashing,
- cookie-based affinity,
- consistent hashing on identifiers.

23.2.12 Port exhaustion and connection limits

Production systems with high concurrency require careful planning:

- connection limits per process and per port,
- OS tuning for maximum file descriptors,
- backlog queue sizing (listen queue),
- delays in TIME_WAIT states can temporarily reduce available sockets.

Key OS knobs (Linux):

- `net.core.somaxconn` (maximum listen queue),
- `net.ipv4.ip_local_port_range` (ephemeral port range),
- `fs.file-max` (file descriptor limit).

23.2.13 Practical network configuration for Wt with reverse proxy

A typical production layout:

- Reverse proxy (e.g., Nginx) binds to ports 80 and 443.
- Proxy terminates TLS and forwards to Wt on an internal port (e.g., 8080).
- Proxy handles routing, compression, caching, security headers.
- Wt sees only HTTP traffic on localhost or a private network.
- Firewall allows only proxy ports from external networks.
- Monitoring systems probe health endpoints via the proxy.

23.2.14 Checklist: networking readiness for Wt on Linux

- Bind Wt only to necessary interfaces (prefer loopback/private IP behind a proxy).
- Use reverse proxy for ports 80/443 and TLS termination.
- Configure firewall to only expose intended ports.
- Plan ephemeral port range for heavy outbound connections.
- Support both IPv4 and IPv6 where needed.
- Implement session affinity or external session storage for scaling out.
- Monitor connection counts, backlog queues, and resource limits.
- Validate health check access patterns and restrict debug endpoints.

23.2.15 Outcome

Understanding and correctly configuring ports and networking fundamentals ensures that a Wt application is reachable, secure, and scalable in production. Correct binding, reverse proxy integration, firewall policy, and load balancer configuration protect the server from unintended exposure while enabling robust service delivery under real-world traffic.

Nginx and Reverse Proxy Setup

24.1 Nginx configuration

In production deployments, Nginx is commonly placed in front of a Wt standalone server as a reverse proxy. This architecture separates responsibilities cleanly:

- **Nginx at the edge**: TLS termination, HTTP/2, security headers, buffering, static file delivery, request limits, rate limiting, and routing.
- **Wt behind the edge**: session-based server-side UI, WebSocket (optional), application routing, business logic, database access.

This section provides a professional Nginx configuration model designed specifically for Wt applications, including:

- safe port exposure and binding strategy,
- correct reverse proxy headers,
- WebSocket upgrade support,
- timeouts for long-lived interactive sessions,
- upload/download considerations,

- static resource handling and caching,
- security hardening at the proxy boundary,
- scaling patterns (multiple instances, session affinity).

24.1.1 Recommended topology

A production-grade topology on Linux is:

- Nginx listens publicly on `443` (and optionally `80` for redirect).
- Wt listens privately on `127.0.0.1:8080` (or a private VLAN IP).
- Firewall blocks direct access to the Wt port from untrusted networks.

This keeps your application server unreachable from the public internet and centralizes HTTPS and security policy enforcement at the reverse proxy.

24.1.2 Core reverse-proxy rules for Wt

A correct reverse proxy for Wt must do these things reliably:

- Preserve the original `Host` header (virtual host correctness).
- Forward scheme information (`https`) so the app can generate correct absolute URLs and secure cookies.
- Forward the client IP chain for logging/rate limiting (`X-Forwarded-For`) *only* from trusted proxy contexts.
- Support WebSocket upgrade if your Wt setup uses WebSockets.
- Set sensible timeouts for interactive, long-lived connections.

24.1.3 A production baseline configuration (HTTP redirect + HTTPS reverse proxy)

The following is a strong baseline. It assumes:

- Wt is listening on `127.0.0.1:8080`.
- TLS is terminated in Nginx.
- You want HTTP redirected to HTTPS.
- WebSocket upgrades are enabled.

```
# ------------------------------------------------------------
# /etc/nginx/conf.d/wt_app.conf (example)
# ------------------------------------------------------------

# 1) Helper map for WebSocket Connection header correctness
map $http_upgrade $connection_upgrade {
  default upgrade;
  ''      close;
}

# 2) Upstream to Wt (single instance; see scaling section for multiple servers)
upstream wt_backend {
  server 127.0.0.1:8080;
  keepalive 32;   # allow upstream keepalive for performance
}

# 3) HTTP -> HTTPS redirect
server {
  listen 80;
  server_name example.com;
```

```
  # Production redirect: all traffic to HTTPS
  return 301 https://$host$request_uri;
}

# 4) HTTPS reverse proxy
server {
  listen 443 ssl http2;
  server_name example.com;

  # --- TLS certificate paths (placeholders) ---
  ssl_certificate     /etc/ssl/example/fullchain.pem;
  ssl_certificate_key /etc/ssl/example/privkey.pem;

  # --- TLS policy (keep modern; adjust to your compatibility needs) ---
  ssl_protocols TLSv1.2 TLSv1.3;
  ssl_session_cache shared:SSL:20m;
  ssl_session_timeout 10m;

  # --- Security headers (baseline) ---
  add_header X-Content-Type-Options "nosniff" always;
  add_header Referrer-Policy "strict-origin-when-cross-origin" always;
  add_header X-Frame-Options "DENY" always;

  # HSTS: enable only after verifying HTTPS works for your host/subdomains
  add_header Strict-Transport-Security "max-age=31536000" always;

  # --- Upload limits (adjust for your app) ---
  client_max_body_size 50m;

  # --- Proxy buffering policy ---
  # Wt apps are interactive; buffering can be tuned.
  proxy_buffering on;
  proxy_buffers 16 16k;
```

```
  proxy_busy_buffers_size 64k;

  # --- Timeouts for long-lived interactive sessions ---
  proxy_connect_timeout 5s;
  proxy_send_timeout    60s;
  proxy_read_timeout    60s;

  # If you use WebSockets and long-lived connections, increase read timeout:
  # proxy_read_timeout  3600s;

  # --- Main application proxy ---
  location / {
    proxy_pass http://wt_backend;

    # Preserve original host and forwarding information
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    # WebSocket upgrade (safe even when requests are not websockets)
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;

    # Upstream keepalive correctness
    proxy_set_header Connection "";

    # If your app sets cookies and you want extra safety, you may enforce:
    # proxy_cookie_path / "/; Secure; HttpOnly; SameSite=Lax";
  }
}
```

24.1.4 Important details and why they matter

Preserving `Host` and scheme

Wt applications may generate links and redirects. If the proxy does not forward:

- `Host`,
- and the original scheme via `X-Forwarded-Proto`,

the application may generate incorrect URLs, incorrect redirects, or fail to set secure cookies consistently.

WebSocket upgrades

If Wt uses WebSockets, Nginx must support:

- HTTP/1.1 to the upstream,
- `Upgrade` and `Connection` headers.

Without this, WebSocket connections fail and the application may degrade to long polling or become unstable.

Timeout tuning for interactive apps

Interactive applications can keep connections open longer than typical REST APIs. Production tuning requires:

- bounded timeouts to avoid resource exhaustion,
- longer timeouts when using WebSockets or server push,
- balanced values that match your expected user activity patterns.

24.1.5 Serving static files efficiently (recommended)

Let Nginx serve static assets directly whenever feasible (icons, CSS, images, downloadable public assets). This reduces load on Wt and improves latency.

```
# Example: serve static files from /var/www/myapp/static/
location /static/ {
  alias /var/www/myapp/static/;
  access_log off;

  # Cache policy for versioned assets (recommended)
  expires 30d;
  add_header Cache-Control "public, max-age=2592000, immutable";
}
```

Operational rule:

- For **versioned** assets (hash in filename), use long caching (`immutable`).
- For **non-versioned** assets, use shorter caching and rely on validation headers.

24.1.6 Rate limiting and request hardening

Reverse proxies are an effective place to reduce abusive traffic. Typical controls:

- rate-limit login attempts,
- limit burst traffic to expensive endpoints,
- cap request body size and header sizes (within reason).

```
# Limit requests per IP (example)
limit_req_zone $binary_remote_addr zone=req_per_ip:10m rate=10r/s;
```

```
server {
  # ...
  location /login {
    limit_req zone=req_per_ip burst=20 nodelay;
    proxy_pass http://wt_backend;
  }
}
```

Use rate limiting carefully:

- Prefer applying it to high-risk endpoints (login, password reset).
- Avoid breaking legitimate bursty UI behavior; tune burst values and scope.

24.1.7 Uploads and downloads

Uploads require:

- `client_max_body_size` sized appropriately,
- timeouts that allow slow clients to finish uploads,
- buffering policy decisions (buffer vs stream).

If you expect large uploads, consider:

- raising `proxy_read_timeout` / `proxy_send_timeout`,
- controlling upload routes explicitly (e.g., `/upload`),
- restricting allowed content types and enforcing application-side validation regardless of proxy checks.

For downloads:

- keep them authorized in the application (do not expose private files via direct Nginx alias),
- consider `X-Accel-Redirect` patterns only when you fully control access decisions and path mapping.

24.1.8 Correct client IP handling behind proxies

If Nginx is the public edge, your application logs and security controls (rate limiting, auditing) often need the real client IP. However, trusting forwarded headers from untrusted sources is dangerous. Production posture:

- Nginx should be the only component that sets the forwarded chain for upstream.
- The Wt application should trust forwarded headers *only* from the known proxy network path.

If you have additional upstream proxies or load balancers, configure Nginx to accept the correct client IP chain only from those trusted sources.

24.1.9 Scaling out: multiple Wt instances and session affinity

Wt applications are often session-stateful. If you run multiple Wt instances behind Nginx, you typically need **session affinity** (sticky sessions) unless session state is externalized.

Upstream with multiple servers

```
upstream wt_backend {
  # Session affinity approach depends on your architecture.
  # IP hash is simple but imperfect for NAT-heavy networks.
  ip_hash;
```

```
  server 10.0.0.11:8080;
  server 10.0.0.12:8080;
  keepalive 64;
}
```

Professional guidance:

- Prefer a load balancer feature that supports cookie-based affinity when available.
- Use IP-based affinity only when acceptable for your user population and network realities.
- If you externalize state (rare for Wt widget trees), you can use pure round-robin load balancing.

24.1.10 Operational hygiene: logs, reloads, and validation

Before reloading Nginx, validate config:

```
nginx -t
```

Reload without dropping connections:

```
systemctl reload nginx
```

Logging posture:

- keep access logs in production but consider sampling for high-volume static paths,
- retain error logs at an appropriate level,
- ensure logs do not store secrets (do not log Authorization headers; avoid logging full request bodies).

24.1.11 A concise production checklist

- Nginx listens on 443 (and 80 for redirect); Wt binds to loopback/private interface.
- Correct proxy headers: Host, X-Forwarded-Proto, X-Forwarded-For.
- WebSocket upgrade enabled (Upgrade/Connection, HTTP/1.1 upstream).
- Timeouts tuned for interactive sessions; not infinite, not too short.
- Static assets served by Nginx with correct caching policy.
- Upload limits set; sensitive endpoints rate-limited.
- Security headers enabled; HSTS rolled out intentionally.
- Firewall blocks direct access to Wt port from the internet.
- Multi-instance deployments use session affinity or an explicitly safe state strategy.

24.1.12 Outcome

A correctly engineered Nginx configuration turns Wt standalone mode into a production-grade deployment: stable TLS termination, consistent security policy enforcement, efficient static delivery, safe WebSocket support, controlled timeouts, and clear scaling options. The result is a Linux deployment model that is predictable, secure, and maintainable under real-world traffic.

24.2 WebSocket forwarding

WebSocket forwarding is the reverse-proxy configuration that allows an external client (the browser) to upgrade an HTTP request into a persistent, full-duplex TCP connection, while Nginx transparently proxies that upgraded connection to a backend server (your Wt standalone

process). For interactive Wt applications, WebSockets are often the preferred transport because they provide:

- low-latency bidirectional messaging for UI events and server push,
- reduced overhead compared to frequent HTTP polling,
- stable long-lived connections that map naturally to session-based server-side UI.

In production, WebSocket forwarding must be engineered carefully. A misconfigured proxy can lead to:

- upgrade failures (WebSocket never establishes),
- intermittent disconnects due to timeouts,
- unexpected connection closures under load,
- incorrect upstream connection reuse behavior,
- broken client IP / scheme detection in the application.

This section provides a professional, production-grade WebSocket forwarding configuration for Nginx in front of a Wt backend, and explains the most important details.

24.2.1 WebSocket upgrade: the essential mechanism

A WebSocket connection begins as an HTTP request containing:

- `Upgrade: websocket`
- `Connection: Upgrade`

If the server accepts, it responds with an HTTP 101 status (Switching Protocols), and the connection becomes a WebSocket stream.
When a reverse proxy sits in between, it must:

- preserve the upgrade headers,
- use HTTP/1.1 to the upstream (WebSocket upgrade is defined for HTTP/1.1),
- keep the connection open long-term,
- avoid buffering behaviors that do not apply to streaming sockets.

24.2.2 Production baseline: Nginx forwarding to Wt

Assumptions:

- Nginx terminates TLS on port 443.
- Wt listens on `127.0.0.1:8080`.
- The reverse proxy forwards both normal HTTP requests and WebSocket upgrades on the same location.

```
# ------------------------------------------------------------
# WebSocket-forwarding baseline for Wt behind Nginx
# ------------------------------------------------------------

# Map Upgrade header to a correct Connection header value.
# This avoids hardcoding "upgrade" and handles non-WebSocket requests safely.
map $http_upgrade $connection_upgrade {
  default upgrade;
  ''      close;
}

upstream wt_backend {
  server 127.0.0.1:8080;
  keepalive 32;
}
```

```
server {
  listen 80;
  server_name example.com;
  return 301 https://$host$request_uri;
}

server {
  listen 443 ssl http2;
  server_name example.com;

  ssl_certificate     /etc/ssl/example/fullchain.pem;
  ssl_certificate_key /etc/ssl/example/privkey.pem;

  ssl_protocols TLSv1.2 TLSv1.3;

  # WebSocket connections are long-lived; choose timeouts deliberately.
  # proxy_read_timeout must exceed expected idle periods or heartbeat intervals.
  proxy_connect_timeout 5s;
  proxy_send_timeout    60s;
  proxy_read_timeout    3600s;  # typical for WS; tune to your app needs

  # Some deployments also tune client-side keepalive:
  keepalive_timeout  65s;

  # Main location: handles normal HTTP and WebSocket upgrade in one place.
  location / {
    proxy_pass http://wt_backend;

    # Preserve original host and scheme for correct absolute URLs and secure cookie logic.
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Proto https;
```

```
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    # REQUIRED for WebSocket: HTTP/1.1 upstream + Upgrade headers.
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;

    # Ensure proxy does not transform streaming behavior unexpectedly.
    proxy_buffering off;

    # Disable caching for upgraded connections; safe for mixed traffic.
    proxy_cache_bypass $http_upgrade;
  }
}
```

24.2.3 Why each directive matters

`proxy_http_version 1.1`

WebSocket upgrade relies on HTTP/1.1 semantics. Even if the browser uses HTTP/2 to Nginx, the upstream hop should be HTTP/1.1 for the upgrade.

`Upgrade` and `Connection` headers

These headers carry the upgrade request through the proxy boundary. Without them, the backend sees a normal HTTP request and will not establish a WebSocket.

`map` for `$connection_upgrade`

Hardcoding `Connection "upgrade"` is a common snippet, but a robust configuration handles both WebSocket and non-WebSocket requests cleanly. Using a `map` ensures:

- WebSocket requests get `Connection: upgrade`,

- normal requests get a safe value (`close`) instead of an invalid upgrade attempt.

`proxy_read_timeout`

WebSockets can sit idle for long periods while the user reads a page. If `proxy_read_timeout` is too short, Nginx will close the connection and the UI will appear to disconnect intermittently. Production tuning should reflect:

- expected user idle time,
- heartbeat/ping interval (if used),
- maximum acceptable resource occupancy per connection.

`proxy_buffering off`

For streaming, buffering is not helpful and can introduce latency or memory overhead. Disabling buffering is a safe default for WebSocket traffic.

24.2.4 Forwarding on a dedicated WebSocket path

Some deployments prefer a dedicated location for WebSockets (clear separation of policy). This is useful if you want:

- different timeouts for WebSockets vs normal HTTP,
- specialized rate limiting,
- separate access logging or monitoring.

```
location /ws/ {
  proxy_pass http://wt_backend;
```

```
  proxy_set_header Host $host;
  proxy_set_header X-Forwarded-Proto https;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

  proxy_http_version 1.1;
  proxy_set_header Upgrade $http_upgrade;
  proxy_set_header Connection $connection_upgrade;

  proxy_buffering off;

  proxy_read_timeout 3600s;
}
```

If you choose this approach, ensure your Wt application is configured to use that path consistently for its WebSocket endpoint.

24.2.5 Client IP and scheme correctness

For operational logging, security auditing, and secure-cookie decisions, the backend must know:

- the original `Host`,
- the external scheme (`https`),
- the client IP chain (for rate-limiting and incident correlation).

Professional rule:

- The backend should trust forwarded headers only from the known proxy network path.
- The Wt port must not be directly reachable from untrusted networks.

24.2.6 Load balancing and session affinity

WebSocket connections are long-lived and typically bound to a single backend instance for their lifetime. For Wt applications that keep server-side session state, scaling out commonly requires **sticky sessions** so subsequent HTTP requests from the same session are routed to the same backend instance.

Simple (but imperfect) affinity approach in Nginx:

```
upstream wt_backend {
  ip_hash;
  server 10.0.0.11:8080;
  server 10.0.0.12:8080;
  keepalive 64;
}
```

Operational notes:

- IP-hash can be unstable behind NAT-heavy networks and mobile carriers.
- Cookie-based affinity (when available in your load balancer stack) is often more stable.
- If a backend restarts, existing WebSocket connections will drop; clients must reconnect gracefully.

24.2.7 Common failure patterns and how to prevent them

Failure: WebSocket never upgrades

Typical causes:

- missing `proxy_http_version 1.1`,
- missing `Upgrade` or `Connection` headers,
- proxy location does not match the WebSocket path.

Failure: Random disconnects after a short period

Typical causes:

- `proxy_read_timeout` too low,
- upstream idle timeouts,
- intermediate firewall/NAT timeouts without keepalive/heartbeats.

Professional mitigations:

- increase `proxy_read_timeout`,
- implement periodic ping/heartbeat at the application level if needed,
- ensure your network path allows long-lived connections.

Failure: Backend sees wrong scheme (http instead of https)

Typical cause:

- missing `X-Forwarded-Proto https`.

Impact:

- incorrect absolute URLs,
- cookies not marked Secure,
- redirect loops or insecure redirects.

Failure: Security controls bypassed by spoofed forwarded headers

Typical cause:

- backend port exposed publicly, allowing attackers to send forged `X-Forwarded-*`.

Mitigation:

- firewall the backend port,
- trust forwarded headers only from the reverse proxy.

24.2.8 Production checklist for WebSocket forwarding

- Nginx terminates TLS; Wt listens on loopback/private interface.
- `proxy_http_version 1.1` is set for WebSocket locations.
- `Upgrade` and `Connection` headers are forwarded using a robust `map`.
- `proxy_read_timeout` is tuned for long-lived idle periods.
- `proxy_buffering off` is enabled for streaming behavior.
- `Host`, `X-Forwarded-Proto`, and `X-Forwarded-For` are set correctly.
- In multi-instance setups, session affinity is planned and tested.
- Monitoring detects disconnect spikes and reconnect storms (symptom of timeout or instability).

24.2.9 Outcome

With correct WebSocket forwarding, Nginx becomes a reliable edge gateway for Wts interactive, session-based model. The result is stable long-lived connections, predictable latency, correct security posture (scheme and client identity), and a reverse-proxy deployment that can be scaled and operated safely on Linux.

24.3 HTTPS integration

HTTPS integration is the disciplined configuration of TLS at the reverse proxy boundary so that every interaction between the browser and your Wt application is encrypted, authenticated, and policy-controlled. In production-grade deployments, HTTPS is not a single switch; it is a system of:

- correct certificate lifecycle management,
- modern TLS protocol and cipher configuration,
- strict redirection and browser enforcement (HSTS),
- secure header policy (CSP, clickjacking, sniffing protections),
- correct upstream forwarding (scheme, client IP),
- and operational controls (timeouts, logging, renewals, reloads).

This section presents a professional HTTPS integration approach for **Nginx in front of a Wt standalone server** on Linux.

24.3.1 Deployment model: TLS terminates at Nginx

The recommended production topology is:

```
Browser (HTTPS :443)  --->  Nginx (TLS termination)  --->  Wt (HTTP :8080 on
↪  localhost/private IP)
```

Key properties:

- Only Nginx is exposed to the public internet.
- Wt binds to `127.0.0.1` (or a private interface) and is firewalled.
- Nginx forwards the original scheme and client context to the backend.

24.3.2 Certificate lifecycle: obtain, renew, reload

HTTPS is operationally safe only if certificate renewal is automated.

Certificate files Nginx typically needs:

- `fullchain.pem`: server certificate plus intermediate chain
- `privkey.pem`: the private key

Automation workflow

- Obtain the certificate using an ACME client.
- Renew periodically via a timer.
- Reload Nginx after renewal without dropping existing connections.

A typical safe reload action:

```
nginx -t && systemctl reload nginx
```

Operational rule:

- Never deploy HTTPS with manual renewal as the primary plan.
- Ensure monitoring alerts on renewal failures or near-expiry.

24.3.3 Baseline HTTPS configuration: redirect + TLS + proxy

The following configuration provides a strong baseline for most Wt deployments.
Assumptions:

- Domain: `example.com`
- Wt backend: `127.0.0.1:8080`
- TLS terminated at Nginx
- WebSockets supported (safe even if your app uses long polling)

```
# ------------------------------------------------------------
# /etc/nginx/conf.d/wt_https.conf (example)
# ------------------------------------------------------------

# WebSocket Connection header helper
map $http_upgrade $connection_upgrade {
  default upgrade;
  ''      close;
}

upstream wt_backend {
  server 127.0.0.1:8080;
  keepalive 32;
}

# 1) HTTP -> HTTPS redirect server
server {
  listen 80;
  server_name example.com;

  # Redirect everything to HTTPS
```

```
  return 301 https://$host$request_uri;
}

# 2) HTTPS termination and reverse proxy server
server {
  listen 443 ssl http2;
  server_name example.com;

  # --- Certificates ---
  ssl_certificate     /etc/ssl/example/fullchain.pem;
  ssl_certificate_key /etc/ssl/example/privkey.pem;

  # --- Protocol policy ---
  ssl_protocols TLSv1.2 TLSv1.3;

  # --- Session resumption (performance) ---
  ssl_session_cache shared:SSL:20m;
  ssl_session_timeout 10m;

  # --- OCSP stapling (optional; requires resolver) ---
  # resolver 1.1.1.1 8.8.8.8 valid=300s;
  # resolver_timeout 5s;
  # ssl_stapling on;
  # ssl_stapling_verify on;

  # --- Security headers baseline ---
  add_header X-Content-Type-Options "nosniff" always;
  add_header X-Frame-Options "DENY" always;
  add_header Referrer-Policy "strict-origin-when-cross-origin" always;

  # HSTS: enable after confirming HTTPS is correct everywhere for this host.
  # Add includeSubDomains only if all subdomains are HTTPS-ready.
  add_header Strict-Transport-Security "max-age=31536000" always;
```

```
  # --- Request size limits (uploads) ---
  client_max_body_size 50m;

  # --- Proxy timeouts (tune for interactive apps / WebSockets) ---
  proxy_connect_timeout 5s;
  proxy_send_timeout    60s;
  proxy_read_timeout    3600s;

  # --- Main proxy location ---
  location / {
    proxy_pass http://wt_backend;

    # Preserve original request context
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Host $host;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    # WebSocket support
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;

    # Streaming behavior: often best for interactive traffic
    proxy_buffering off;
    proxy_cache_bypass $http_upgrade;
  }
}
```

24.3.4 Why each HTTPS component matters

HTTP-to-HTTPS redirection

Redirecting port 80 to 443 is not just convenience:

- prevents accidental login over HTTP,
- normalizes all traffic to secure transport,
- reduces risk of session cookie exposure.

TLS protocol selection

Modern deployments prefer TLS 1.3 and allow TLS 1.2 only for compatibility. Disabling obsolete protocols is mandatory for safe operation.

HSTS

HSTS instructs browsers to never attempt HTTP for the host during the policy window. This prevents protocol downgrade attacks after the first secure visit.
Rollout guidance:

- start with a small `max-age` (e.g., hours or days),
- increase gradually after confirming correct HTTPS behavior,
- avoid `includeSubDomains` unless every subdomain is ready.

Forwarded scheme and secure cookies

When TLS terminates at Nginx, your Wt backend receives plain HTTP. Without `X-Forwarded-Proto: https`, the application may:

- generate incorrect redirects,
- produce absolute URLs with `http`,
- fail to consistently set Secure cookies depending on logic.

24.3.5 Cookie and session security coupling with HTTPS

HTTPS protects cookies in transit, but you must also harden cookie attributes.
Recommended cookie attributes:

- **Secure**: send only over HTTPS
- **HttpOnly**: not readable by JavaScript
- **SameSite**: reduces CSRF risk

Professional rule:

- Never rely on HTTPS alone to protect sessions; enforce secure cookie policy and robust session management.

24.3.6 Security headers and HTTPS

HTTPS provides encrypted transport, but modern browsers also need policy headers to reduce common attacks.
Baseline headers (already shown):

- `X-Content-Type-Options: nosniff`
- `X-Frame-Options: DENY` (or a precise framing policy if embedding is required)
- `Referrer-Policy: strict-origin-when-cross-origin`

Content Security Policy (CSP) is highly valuable but application-specific. A safe workflow:

- begin with a report-only policy,
- observe violations,
- tighten gradually until enforcement is safe.

Example (illustrative; requires tailoring):

```
# add_header Content-Security-Policy "default-src 'self'; frame-ancestors 'none'; base-uri
↪  'self'" always;
```

24.3.7 WebSockets over HTTPS: WSS

When the site is HTTPS, WebSockets should run as WSS. Nginx configuration must:

- support the Upgrade mechanism,
- keep connections open long enough,
- avoid proxy buffering for streaming behavior.

If you observe frequent disconnects:

- increase `proxy_read_timeout`,
- ensure your network path does not kill idle TCP connections too aggressively,
- consider application-level heartbeats (bounded and rate-limited).

24.3.8 Uploads, buffering, and HTTPS

HTTPS affects uploads because the proxy now decrypts and may buffer request bodies. Key controls:

- `client_max_body_size` prevents accidental or malicious oversized uploads.
- timeouts must allow slow clients but remain bounded.
- buffering can be tuned; for large uploads, consider explicit upload locations with specialized timeouts.

24.3.9 Correct client IP behavior behind TLS termination

When Nginx is the public edge, the backend sees Nginx as the TCP peer. To preserve the real client IP in logs and security controls:

- forward `X-Forwarded-For`,
- ensure the Wt backend trusts forwarded headers only from Nginx,
- firewall the backend so it is not directly reachable from the internet.

24.3.10 Operational hardening checklist for HTTPS

- Certificates are automated (ACME) and renewed with a reload hook.
- HTTP (80) permanently redirects to HTTPS (443).
- TLS protocols restricted to modern versions (TLS 1.3 with TLS 1.2 fallback if needed).
- HSTS enabled with a safe rollout strategy.
- Forwarded headers set correctly: Host, X-Forwarded-Proto, X-Forwarded-For.

- Secure header baseline applied; CSP introduced progressively if feasible.
- WebSockets are forwarded correctly (Upgrade/Connection, HTTP/1.1 upstream).
- Upload limits and timeouts are defined and tested.
- Backend Wt port is bound to loopback/private interface and blocked from the public internet.
- Logging avoids secrets (no tokens, no passwords, no raw session IDs).

24.3.11 Outcome

With HTTPS integrated at Nginx, a Wt deployment gains a production-grade security boundary: encrypted transport, verified server identity, browser-enforced HTTPS behavior, consistent security headers, and a clean separation between edge responsibilities and application responsibilities. This integration is the standard foundation for secure, stable Wt deployments on Linux.

systemd and Professional Service Management

25.1 Creating a systemd service

In production deployments on Linux, services must be managed reliably. The de facto standard on modern Linux distributions is **systemd**, a service manager that provides:

- process supervision,
- restart policies,
- dependency ordering,
- logging integration,
- resource limits,
- and secure sandboxing.

A Wt standalone application (or any backend process) should run under `systemd` so that it starts automatically on boot, restarts on failure, and is observable through standard tooling. This section describes how to design, implement, and refine a professional `systemd` service unit for a Wt application on Linux.

25.1.1 Service unit structure

A minimal `systemd` service unit has three top-level sections:

- `[Unit]` metadata and ordering.
- `[Service]` how the process is executed and supervised.
- `[Install]` how the service is enabled/started by default.

25.1.2 Example service unit for a Wt application

Suppose your Wt standalone server binary is `/opt/myapp/bin/wtapp` and your configuration file resides at `/etc/myapp/wt_config.xml`. A production service unit might be:

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Wt Standalone Application
After=network.target

[Service]
Type=simple
User=wtapp
Group=wtapp
WorkingDirectory=/opt/myapp

# The command line for starting the service; adjust argv as needed.
ExecStart=/opt/myapp/bin/wtapp --config /etc/myapp/wt_config.xml

# Restart on failure (transient and non-zero exit codes).
Restart=on-failure
RestartSec=5s

# Resource safety: do not allow the service to spawn new privileges.
```

```
NoNewPrivileges=true

# Time and file hardening (reduce attack surface).
PrivateTmp=true
ProtectSystem=full
ProtectHome=true

# Specify directories the service may write to.
ReadWritePaths=/var/log/myapp /var/lib/myapp

# Logging goes to the journal by default; you can redirect if desired.
StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

25.1.3 Key directives explained

Unit section

- `Description` human-readable summary.
- `After` ensures this service starts after fundamental system dependencies (e.g., `network.target`).

Service section: execution policy

- `Type=simple` the default; assumes `ExecStart` is the main process.
- `User` and `Group` run the service as an unprivileged account dedicated to this application.
- `WorkingDirectory` set the CWD for relative paths used by the service.

- `ExecStart` invoked with full path and parameters; do not use shell syntax.
- `Restart=on-failure` automatically restarts on non-zero exits or crashes.
- `RestartSec` wait before restarting to avoid tight crash loops.

Service section: security hardening

- `NoNewPrivileges=true` prevents privilege escalation via execve() after start.
- `PrivateTmp=true` provides private `/tmp` and `/var/tmp` namespaces.
- `ProtectSystem` restricts access to read-only system paths; `full` protects `/usr` and `/boot`.
- `ProtectHome` restricts access to user home directories.
- `ReadWritePaths` explicitly allow writeable paths for logging and state.
- `StandardOutput` and `StandardError` send output to the system journal for centralized collection.

Install section

- `WantedBy=multi-user.target` makes the service start automatically on multi-user (non-graphical) boot.

25.1.4 Creating a dedicated service user

For security, do not run application services as `root`. Create a dedicated user and group:

```
useradd --system --no-create-home --shell /usr/sbin/nologin wtapp
mkdir -p /var/log/myapp
mkdir -p /var/lib/myapp
chown -R wtapp:wtapp /var/log/myapp /var/lib/myapp
```

This ensures:

- The service cannot modify unrelated system files.
- File ownership matches service expectations.
- Auditing and accountability is clear.

25.1.5 Enabling and managing the service

Once the unit file is in place:

```
# Reload systemd to recognize the new unit
systemctl daemon-reload

# Enable the service to start at boot
systemctl enable myapp.service

# Start the service now
systemctl start myapp.service

# Check status and logs
systemctl status myapp.service
journalctl -u myapp.service --no-pager
```

25.1.6 Handling configuration changes

When modifying the service unit or related configuration (e.g., `wt_config.xml`), use:

```
systemctl daemon-reload
systemctl restart myapp.service
```

This ensures:

- systemd picks up unit changes,
- service restarts cleanly under supervision.

25.1.7 Resource limits and service tuning

You can set explicit limits for production robustness:

```
# Inside [Service]
LimitNOFILE=65536        # allow many file descriptors
LimitNPROC=4096          # process count limit
TimeoutStartSec=30s      # how long to wait for start
TimeoutStopSec=30s       # how long to wait for stop
```

File descriptor limits matter for high concurrency; Wt applications handling many sockets or files should set `LimitNOFILE` accordingly.

Timeouts ensure systemd does not consider a slow start or stop as failed prematurely.

25.1.8 Logging and observability

By default, stdout and stderr go to the system journal. For structured logging and retention:

- Use `journalctl` with filters (`-u myapp.service`) to inspect logs.
- Integrate with centralized logging systems via the journal (e.g., forwarding to ELK/EFK or a log aggregator).
- Avoid logging secrets or raw tokens in application output.

If you prefer log files instead of (or in addition to) the journal, you can redirect output:

```
# In [Service]
StandardOutput=append:/var/log/myapp/output.log
StandardError=append:/var/log/myapp/error.log
```

Be mindful:

- rotation must be handled (e.g., via `logrotate`),
- permissions must allow the service to write.

25.1.9 Graceful shutdown and signals

systemd sends signals to the service:

- `SIGTERM` for graceful shutdown,
- after `TimeoutStopSec`, `SIGKILL` if the process did not exit.

Your application should:

- respond to `SIGTERM` by shutting down cleanly,
- close sockets and free resources,
- flush logs and state to durable storage.

If your Wt app blocks on long operations, ensure you implement cancellation or timeout logic that cooperates with systemd-initiated shutdown.

25.1.10 Security hardening considerations

To reduce attack surface further:

- Use `AmbientCapabilities=` sparingly or not at all.

- Use `CapabilityBoundingSet=` to drop unnecessary capabilities.
- Use `ProtectKernelTunables=true` and `ProtectControlGroups=true` when appropriate.
- Consider `PrivateDevices=true` if the service does not need device access.
- Use `SystemCallFilter` to restrict allowed syscalls if your deployment has strict security requirements.

Example syscall filter (very restrictive; adjust carefully):

```
# In [Service]
SystemCallFilter=@default @networking
```

25.1.11 High-availability considerations

For mission-critical deployments:

- Place the service behind a load balancer or proxy that can detect unhealthy instances.
- Use health checks (HTTP or metrics endpoint) polled by the load balancer.
- Consider active/passive redundancy and rolling updates coordinated with `systemd` and orchestration tooling.

25.1.12 Monitoring systemd services

Monitor service health via:

```
systemctl is-active myapp.service
systemctl is-failed myapp.service
journalctl -f -u myapp.service
```

For proactive alerting:

- integrate systemd state with monitoring tools (Prometheus node exporter / systemd exporter),
- trigger alerts on repeated restarts or non-active states,
- correlate service logs with application logs for incidents.

25.1.13 Outcome

A professionally configured `systemd` service ensures that your Wt application:

- starts automatically at boot,
- restarts on failures,
- runs with least privilege,
- is observable via standard tooling,
- and is hardened against common attack vectors.

Combined with appropriate logging, monitoring, and TLS integration at the reverse proxy boundary, this forms the backbone of a stable, secure, and maintainable Linux production deployment.

25.2 Automatic restarts

Automatic restarts are a core reliability feature of `systemd`. In production, processes can terminate due to:

- crashes (segmentation faults, aborts),

- resource exhaustion (OOM kills),
- transient dependency failures that trigger fatal exits,
- misconfiguration during rollout,
- operator actions (manual stop/start sequences).

A professionally managed Wt server should be configured so that:

- the service restarts when it fails unexpectedly,
- restart behavior is controlled to avoid restart storms,
- failure signals are observable and alertable,
- repeated failures trigger protective behavior rather than endless loops.

This section explains how to design restart policies in `systemd` for a Wt standalone server on Linux.

25.2.1 What systemd considers a failure

`systemd` marks a service start attempt as failed when the main process:

- exits with a non-zero exit code,
- terminates due to a signal (e.g., `SIGSEGV`, `SIGABRT`),
- fails to start within `TimeoutStartSec`,
- violates configured runtime limits (resource, watchdog),
- or fails according to unit conditions (e.g., `ExecStartPre` failures).

This definition matters because restart behavior is triggered by failure classification, not by the presence of errors in logs.

25.2.2 Core restart directives

The restart policy is controlled mainly by:

- `Restart=` (when to restart),
- `RestartSec=` (delay before restarting),
- and optional rate limiting directives (`StartLimitIntervalSec=` and `StartLimitBurst=`).

Restart modes

`Restart=on-failure` Restart only when the service fails (non-zero exit or signal). This is the most common production setting for application servers.

`Restart=always` Restart regardless of exit reason, including clean exit. Use this only if your service is designed to never exit under normal conditions.

`Restart=on-abnormal` Restart on abnormal termination (signals, timeouts), but not on non-zero exit codes. Rarely the best choice for application servers.

`Restart=on-success` Restart only when the service exits successfully. Useful for jobs, not for servers.
For Wt servers, the typical professional default is:

- `Restart=on-failure`

because it restarts on crashes and unexpected exits, but not when an operator intentionally stops the service.

25.2.3 A professional baseline restart configuration

```
# In /etc/systemd/system/myapp.service

[Service]
Type=simple
ExecStart=/opt/myapp/bin/wtapp --config /etc/myapp/wt_config.xml

# Restart policy
Restart=on-failure
RestartSec=5s

# Optional: treat some exit codes as "success" to avoid restart loops
# when your program intentionally exits with a known code.
# SuccessExitStatus=2 3

# Optional: prevent restart storms (unit-level, see [Unit] section below)
```

Why `RestartSec` matters Without a delay, a crashing process can restart immediately in a tight loop, causing:

- CPU waste and log floods,
- repeated dependency pressure (DB reconnection storms),
- denial-of-service against itself,
- and noisy operational alerts.

A modest delay (e.g., 2–10 seconds) is a safe baseline. For services that depend on external systems, a longer delay can reduce pressure.

25.2.4 Rate limiting: preventing restart storms

`systemd` includes start-rate limiting to prevent infinite restart loops from hammering the machine.
Common directives:

- `StartLimitIntervalSec=` sets the window in which starts are counted.
- `StartLimitBurst=` sets how many starts are allowed in that window.

If the limit is exceeded, systemd marks the unit as failed and stops restarting until manual intervention or until the interval passes (depending on policy).

```
# In /etc/systemd/system/myapp.service

[Unit]
StartLimitIntervalSec=60
StartLimitBurst=5

[Service]
Restart=on-failure
RestartSec=5s
```

Interpretation:

- If the service restarts more than 5 times within 60 seconds, it will stop restarting and remain in a failed state.

This is critical for production stability. It ensures that repeated crashes trigger a visible failure state rather than an endless loop.

25.2.5 Distinguishing expected exits from failures

Sometimes a service exits intentionally:

- a configuration validation step fails early with a known exit code,
- an operator triggers a controlled one-shot mode,
- a maintenance action ends the process cleanly.

If your application uses a specific exit code to indicate a non-failure condition, you can teach systemd to treat it as success:

```
[Service]
Restart=on-failure
SuccessExitStatus=2 3
```

Now exit codes 2 and 3 are considered successful and will not trigger an automatic restart.

25.2.6 Choosing the correct `Type=` and its impact on restarts

For most Wt standalone servers:

- `Type=simple` is appropriate (the process stays in the foreground).

If you use `Type=forking`, systemd must track the correct PID. Misconfiguration can cause:

- false positives (systemd thinks the service exited),
- incorrect restart behavior,
- orphaned processes.

Professional rule:

- Prefer non-forking services for modern C++ servers and let systemd supervise the main process directly.

25.2.7 Handling OOM kills and memory-related restarts

If the kernel kills your process due to out-of-memory, systemd can restart it, but this is only a symptom mitigation. Production-grade handling includes:

- memory limits to contain impact,
- monitoring RSS trends to detect leaks,
- reducing per-session memory footprint,
- controlling caches and uploads,
- load shedding or backpressure under high load.

Containment controls (examples)

```
[Service]
# Limit memory usage (choose values that match your environment)
MemoryMax=2G

# Optional: accounting for resource visibility
MemoryAccounting=true
CPUAccounting=true
```

If a memory limit is hit, systemd may stop or kill the process depending on configuration and kernel behavior. The restart policy then determines whether it is restarted.

25.2.8 Restart behavior and dependencies

Automatic restarts interact with dependencies such as databases and caches. Restart storms can amplify outages by:

- generating repeated connection bursts,

- exhausting DB connection limits,
- triggering rate-limits upstream.

Professional mitigations:

- use exponential backoff logic inside the application when dependencies are down,
- set `RestartSec` to a non-trivial value,
- consider adding an `ExecStartPre` dependency check *only if* it is bounded by strict timeouts.

Example bounded pre-check (conceptual):

```
[Service]
ExecStartPre=/usr/bin/bash -c 'timeout 2 bash -c "</dev/tcp/127.0.0.1/5432"'
ExecStart=/opt/myapp/bin/wtapp --config /etc/myapp/wt_config.xml
Restart=on-failure
RestartSec=5s
```

Use with caution: blocking pre-checks can delay startup and may not reflect true readiness. Prefer application-level readiness endpoints and dependency-aware retry logic.

25.2.9 Observability of restarts

In production, restarts must be visible:

- systemd logs each start/stop and failure reason in the journal,
- repeated restarts should trigger alerts,
- application logs should include a startup banner with build ID and configuration summary (non-secret).

Useful commands:

```
systemctl status myapp.service
journalctl -u myapp.service --no-pager
journalctl -u myapp.service -f
```

Operational rule:

- Treat unexpected restarts as incidents to investigate, not as normal behavior.

25.2.10 Recommended restart policy for Wt servers

A practical production profile for a Wt standalone server behind Nginx:

```
# /etc/systemd/system/myapp.service
[Unit]
Description=My Wt Server
After=network.target
StartLimitIntervalSec=60
StartLimitBurst=5

[Service]
Type=simple
User=wtapp
Group=wtapp
WorkingDirectory=/opt/myapp
ExecStart=/opt/myapp/bin/wtapp --config /etc/myapp/wt_config.xml

Restart=on-failure
RestartSec=5s

# Useful for high concurrency
LimitNOFILE=65536
```

```
# Hardening
NoNewPrivileges=true
PrivateTmp=true
ProtectSystem=full
ProtectHome=true
ReadWritePaths=/var/log/myapp /var/lib/myapp

StandardOutput=journal
StandardError=journal

[Install]
WantedBy=multi-user.target
```

25.2.11 Checklist: automatic restarts done professionally

- Use `Restart=on-failure` for servers.
- Always set `RestartSec` to avoid tight loops.
- Configure rate limiting (`StartLimitIntervalSec`, `StartLimitBurst`).
- Ensure the service runs in the foreground (`Type=simple`) so systemd supervises correctly.
- Treat repeated restarts as a signal; alert on them.
- Avoid using restarts to hide systemic issues (memory leaks, persistent dependency failure).
- Keep dependency behavior bounded (timeouts/backoff) to prevent reconnection storms.

25.2.12 Outcome

With a well-designed restart policy, systemd becomes a reliable safety net for your Wt server: it recovers automatically from crashes and transient failures, avoids restart storms through rate limiting, and makes failures visible through standard tooling. This is a fundamental building block of professional service management on Linux.

25.3 Basic monitoring

Basic monitoring is the first layer of ensuring that a production Wt deployment remains healthy, responsive, and operational under real traffic. It is not sufficient to simply deploy a service; you must also continuously observe key indicators of health and performance, detect anomalies early, and alert appropriately. Basic monitoring covers systemd service state, resource utilization, latency and error rates, dependency availability, and session health. This section describes foundational monitoring concepts and practical methods suitable for a Wt application managed by systemd on Linux, with an emphasis on actionable signals and integration with standard tooling.

25.3.1 Monitoring goals

Monitoring should make it possible to:

- detect failures and degraded service quickly,
- measure performance trends (latency, throughput),
- understand resource usage (CPU, memory, disk),
- identify dependency issues (database connectivity, storage availability),
- correlate events across the service stack.

A simple deployment can begin with system and service metrics, and then graduate to more advanced observability as needed.

25.3.2 Systemd service health

At the most basic level, monitoring the systemd unit that runs your Wt server provides insight into whether the service is running or has failed. Relevant metrics include:

- active/inactive state,
- restart count,
- last exit code or signal,
- uptime duration,
- failure reasons logged in the journal.

Administrators can use built-in systemd commands for manual inspection, but automated monitoring systems should periodically query the service state:

```
systemctl is-active myapp.service
systemctl show -p SubState,ActiveState myapp.service
```

Alert on non-`active` or `failed` states.

25.3.3 Resource utilization monitoring

Production deployments must be observant of system resource usage to detect saturation, leaks, or contention.

CPU utilization

Track:

- overall CPU load,
- per-process usage for the Wt server,
- thread pool saturation signals.

High sustained CPU usage may indicate:

- inefficient code paths,
- excessive polling loops,
- unbounded request load without queuing.

Memory usage

Monitor:

- resident set size (RSS),
- heap usage if available via allocator introspection,
- fragmentation indications,
- per-session memory footprint if trackable.

Gradual increases in RSS without corresponding workload changes often indicate memory leaks.

File descriptor and socket counts

A typical Wt process uses many sockets and file handles. Monitor:

- file descriptor count,
- socket states (e.g., many `TIME_WAIT` or `CLOSE_WAIT`),
- limits approaching `LimitNOFILE`.

Use:

```
lsof -p $(pidof myapp) | wc -l
```

25.3.4 Basic health endpoints

While systemd reflects whether the process is running, application-level health endpoints provide finer-grained signals:

- **Liveness** the application is able to respond meaningfully.
- **Readiness** the application is ready to handle real traffic (dependencies healthy, migrations applied).

A simple HTTP handler (or dedicated lightweight TCP responder) that returns success only when critical dependencies are healthy is a basic diagnostic.

25.3.5 Latency and error monitoring

Tracking request latency and error rates helps detect performance regressions and service issues before they escalate into outages.
Track:

- p50/p95/p99 latency percentiles,

- request rate (RPS),
- error rate (4xx/5xx rates),
- timeouts and retries.

These can be obtained via:

- reverse proxy logs (Nginx access logs),
- application logs with structured fields,
- metrics endpoints integrated with collectors (Prometheus, etc.).

25.3.6 Dependency availability checks

Dependencies such as:

- databases (PostgreSQL, MySQL),
- cache layers (Redis, Memcached),
- external APIs,
- object storage,

should be monitored. Basic checks include:

- connectivity checks (TCP/health endpoints),
- latency and error distribution,
- pool exhaustion signals (connection pool waits).

Failures in dependencies often precede application-level errors.

25.3.7 Logs as a monitoring source

Application and system logs provide raw signals for issues not captured by simple metrics:

- unexpected exceptions,
- repeated authorization failures,
- upload errors,
- session creation spikes,
- repeated restarts or crash loops.

Centralize logs and apply:

- alert rules on error keywords,
- rate-limited patterns to avoid noise,
- grouping by request ID or session tags for correlation.

25.3.8 Example: simple Nginx access log analysis

If using Nginx in front of Wt, access logs provide a first view of:

- request latency via logged timing fields,
- client error and server error rates,
- suspicious activity (headlines such as repeated 401/403),
- rate spikes that may indicate load issues.

Structuring Nginx access logs with defined fields (remote IP, status, bytes sent, request time) makes automated analysis easier.

25.3.9 Alerting on thresholds

Define alert thresholds that reflect realistic expectations:

- service not running,
- sustained high CPU (>80% for a period),
- memory growth beyond expected bounds,
- error rate above a defined percentage of total requests,
- latency degradation (p95 above SLA target),
- dependency connectivity failures.

Alerts should route to appropriate channels (email, SMS, paging, chat ops) with actionable context.

25.3.10 Monitoring tooling options

Basic monitoring can be implemented with:

- system utilities (`top`, `vmstat`, `ss`, `iostat`),
- reverse proxy logs and analysis scripts,
- systemd timers and service state checks,
- metrics collectors (Prometheus node exporter, custom metrics endpoint),
- external uptime monitors for the public interface.

25.3.11 Metrics collection and exporters

A minimally invasive metrics pipeline can use:

- node exporters for system metrics,
- application metrics exposed via a lightweight endpoint (e.g., `/metrics`),
- collectors that scrape and store metrics for dashboards and alerts.

Even without a full observability stack, collecting basic counts and histograms enables trend detection.

25.3.12 Health checks integrated with orchestration

If the deployment is orchestrated (e.g., Kubernetes or service mesh), the health endpoints discussed earlier feed into orchestrator health checks. For systemd-based deployments:

- simple HTTP checks from a monitoring host at regular intervals,
- internal health scripts that run as timers and report findings.

Correlate health check failures with systemd service state to avoid alerting on planned service restarts or maintenance windows.

25.3.13 Resource limit proximity monitoring

Monitor kernel and service-level limits:

- open file descriptors approaching `LimitNOFILE`,
- thread counts near configured limits,
- memory reservation approaching configured caps.

Proactive alerts on nearing limits prevent outages due to resource exhaustion.

25.3.14 Combining metrics and logs for richer signals

Correlate metrics with logs:

- spikes in latency with specific error messages,
- high session churn with session creation logs,
- dependency timeout spikes with downstream errors.

This correlation improves signal-to-noise ratio and avoids chasing misleading alerts.

25.3.15 Session-level monitoring signals

Session-based frameworks like Wt benefit from session-level signals:

- active session count,
- idle session count,
- session event rates,
- session timeout rates.

High idle sessions with low event activity may indicate abandoned user flows or improper timeout configuration.

25.3.16 Alert automation and escalation policies

Define clear policies:

- who gets notified for what classes of alerts,
- how alerts escalate over time,

- on-call rotations,
- blackout windows for maintenance.

Avoid alert fatigue by tuning thresholds and noise patterns.

25.3.17 Documentation and runbooks

Basic monitoring must be coupled with operational documentation:

- how to interpret key metrics,
- how to access relevant dashboards and logs,
- runbooks for common failures (dependency outage, memory leak spike),
- escalation procedures and contacts.

25.3.18 Outcome

Basic monitoring provides the foundational signals that keep a Wt-based service healthy in production. By tracking service state, resource usage, latency and error rates, dependency connectivity, and session health, engineers gain early warning of operational issues. Coupled with alerts and documentation, basic monitoring enables scalable and reliable operations even before advanced observability (distributed tracing, application performance monitoring) is introduced. The goal is actionable insight, predictable performance, and fast detection of emerging problems.

Mastery and Scalability

Configuration Management

27.1 Configuration files vs environment variables

Configuration management is a reliability and scalability discipline: it determines how your application behaves across environments (development, staging, production), how safely you can deploy changes, and how quickly you can recover from incidents. In production-grade systems, configuration is not "just settings"; it is an operational interface that must be:

- predictable (clear precedence and defaults),
- verifiable (validated at startup),
- secure (secrets are handled correctly),
- observable (effective config can be inspected safely),
- reproducible (deployments can be recreated exactly).

This section compares two dominant mechanisms for application configuration in Linux deployments—**configuration files** and **environment variables**—and then presents a professional model that combines both with explicit precedence, strong validation, and safe secret handling for Wt-based server applications.

27.1.1 Definitions and scope

Configuration file A file read at startup (or reload time) that contains key/value settings, often structured (INI, JSON, YAML, TOML, XML). It may be managed by configuration management tools, packaged with the application, or maintained under `/etc` in Linux.

Environment variable A key/value string in the process environment, supplied by the service manager (systemd), container runtime, shell, or orchestration platform. Environment variables are inherited by child processes and are accessible via `getenv()`.

Hard rule

- **Configuration** defines behavior and connectivity.
- **Secrets** (passwords, private keys, tokens) require additional controls and should not be treated as ordinary config.

27.1.2 Why this choice matters in scalable systems

At small scale, almost any method works. At large scale, the differences become operationally decisive:

- How quickly can you deploy configuration changes?
- How do you prevent configuration drift between servers?
- Can you roll back safely?
- Can you audit changes and prove what was active at time of incident?
- Can you rotate secrets without leaking them into logs and tooling?

A correct configuration strategy reduces outages more than many “performance optimizations” ever will.

27.1.3 Configuration files: strengths and weaknesses

Strengths of configuration files

1) Structure, schema, and readability Files can express complex configuration with nested structure:

- multiple databases,
- per-module logging rules,
- rate limits per endpoint,
- multi-tenant routing,
- feature flags with groups.

2) Natural fit to Linux conventions Linux server configuration traditionally lives in `/etc/<app>/`. Operators expect:

- stable paths,
- permissions controlled by the filesystem,
- backups and diff tools,
- human review and change control.

3) Better for large sets of non-secret settings Many settings are not sensitive but numerous:

- cache sizes,
- thresholds,
- feature toggles,

- timeouts,
- UI behavior parameters,
- per-tenant policy values.

4) Auditing and versioning A file can be committed to a repository (for non-secret values), tracked, reviewed, and rolled back.

Weaknesses of configuration files

1) Distribution and drift If multiple servers each have local files, drift can occur unless managed centrally.

2) Reload complexity Some apps can reload config without restart, others cannot. Reloading safely is non-trivial:

- partial reloads can create inconsistent state,
- some changes require rebuild of pools or listeners,
- bad reloads can destabilize a running process.

3) Secrets-in-files risks If secrets are stored in plain files, you must:

- restrict permissions carefully,
- avoid backups and logs leaking them,
- handle rotations and ownership correctly.

27.1.4 Environment variables: strengths and weaknesses

Strengths of environment variables

1) Excellent for deployment-time overrides Environment variables are a natural mechanism for:

- selecting environment (`APP_ENV=prod`),
- binding addresses and ports,
- toggling features during incident mitigation,
- injecting service discovery endpoints.

2) Fit to systemd and orchestration systemd provides first-class mechanisms to define environment variables in unit files and drop-ins. Orchestrators can inject env vars per deployment without editing files in-place.

3) Minimizes file management in immutable deployments If your deployment artifact is immutable (image or packaged directory), env vars allow configuration without modifying the artifact.

Weaknesses of environment variables

1) Poor structure and ergonomics Environment variables are strings. Complex structured config becomes:

- hard to validate,
- hard to document,
- easy to misquote/escape incorrectly,

- painful to review.

2) Easy to leak unintentionally They can leak through:

- process listings or debugging tools,
- crash reports,
- accidental logging of the environment,
- support bundles.

3) Harder auditing Unless you snapshot them explicitly, it may be unclear what values were active at incident time.

4) Type conversion hazards Everything arrives as text. Mistakes like `TIMEOUT=O` (letter O) can pass silently unless validated.

27.1.5 A professional position: use both, with explicit roles

In mature systems, the best strategy is not choosing one—it is assigning clear responsibilities:

- **Configuration file**: the main source of structured, documented, non-secret settings.
- **Environment variables**: deployment-level overrides and selectors (environment, endpoints, a small set of critical toggles).
- **Secrets**: managed via dedicated secret mechanisms, and only referenced by config (paths, handles), not embedded broadly.

27.1.6 Recommended precedence model

A predictable precedence model eliminates ambiguity and reduces operational mistakes.

1. **Built-in defaults** (safe development defaults; never production secrets).
2. **Configuration file** (primary settings).
3. **Environment variables** (override selected keys).
4. **Command-line arguments** (highest priority; used for emergency overrides and diagnostics).

Make this precedence explicit in documentation and in logs (without printing secrets).

27.1.7 Validation: the non-negotiable requirement

Regardless of source, configuration must be validated **before** the service starts accepting traffic. Validate:

- required fields exist,
- types and ranges are correct (ports 1–65535),
- paths are accessible and permissioned,
- mutually exclusive options are consistent,
- security invariants are enforced (TLS required in production mode),
- dependency endpoints are syntactically valid.

Fail fast:

- If validation fails, exit with a clear error message and non-zero code.
- Do not start “partially” and hope operators notice later.

27.1.8 Secrets: treat them differently

Professional rules for secrets:

- Do not store secrets in source control.
- Do not print secrets in logs or error messages.
- Prefer references to secrets (paths, identifiers) over embedding.
- Prefer rotation-friendly design (reload secrets without redeploy when possible).

A safe pattern:

- Config file defines *where* the secret is (path or secret-provider key).
- The application reads the secret at runtime from a restricted location.

27.1.9 systemd-friendly patterns

systemd supports both approaches well:

- store stable config in `/etc/myapp/config.toml` (or similar),
- use an `EnvironmentFile=` for small runtime selectors,
- use systemd drop-ins for environment-specific overrides,
- store secret files with strict permissions (root-owned, readable by the service user only if needed).

Example: service uses a config file plus an environment file:

```
# /etc/systemd/system/myapp.service
[Service]
ExecStart=/opt/myapp/bin/myapp --config /etc/myapp/config.toml
EnvironmentFile=-/etc/myapp/myapp.env
Restart=on-failure
```

Example environment file (keep it small and non-secret if possible):

```
# /etc/myapp/myapp.env
APP_ENV=production
APP_HTTP_BIND=127.0.0.1
APP_HTTP_PORT=8080
APP_LOG_LEVEL=info
```

27.1.10 Designing the configuration surface for Wt applications

Wt deployments typically include:

- bind address and port (or connector config),
- reverse-proxy awareness (`X-Forwarded-Proto` expectations),
- session timeouts and resource limits,
- upload limits and storage paths,
- logging and diagnostics configuration,
- database connectivity and pool sizing,
- feature flags for UI flows or expensive operations.

Professional advice:

- Keep Wt connector settings in one place (file), not scattered across many env vars.

- Use env vars for a small, controlled override set (bind port, environment, log level).
- Always log a **sanitized config summary** on startup:
  - effective environment,
  - bind address/port,
  - build/version ID,
  - enabled features (names only),
  - dependency hostnames (not credentials).

27.1.11 C++ implementation pattern: typed config + layered sources

A professional C++ configuration subsystem should:

- expose a typed configuration object,
- load file config first,
- apply env overrides,
- apply CLI overrides,
- validate,
- then freeze (immutable during runtime unless you implement explicit reload).

```
#include <cstdlib>
#include <optional>
#include <stdexcept>
#include <string>
#include <string_view>
```

```
static std::optional<std::string> getenv_opt(const char* key) {
  if (const char* v = std::getenv(key)) {
    if (*v != '\0') return std::string(v);
  }
  return std::nullopt;
}

static int parse_int_in_range(std::string_view s, int lo, int hi, const char* name) {
  try {
    size_t idx = 0;
    int v = std::stoi(std::string(s), &idx, 10);
    if (idx != s.size()) throw std::runtime_error("trailing characters");
    if (v < lo || v > hi) throw std::runtime_error("out of range");
    return v;
  } catch (const std::exception&) {
    throw std::runtime_error(std::string("Invalid ") + name);
  }
}

struct AppConfig {
  std::string env = "development";      // dev/staging/production
  std::string http_bind = "127.0.0.1";  // recommended behind Nginx
  int http_port = 8080;
  std::string log_level = "info";       // debug/info/warn/error
  bool require_https = false;           // enforce HTTPS-only semantics in production logic
};

static void apply_env_overrides(AppConfig& c) {
  if (auto v = getenv_opt("APP_ENV"))       c.env = *v;
  if (auto v = getenv_opt("APP_HTTP_BIND")) c.http_bind = *v;
  if (auto v = getenv_opt("APP_HTTP_PORT")) c.http_port = parse_int_in_range(*v, 1, 65535,
  ↪  "APP_HTTP_PORT");
  if (auto v = getenv_opt("APP_LOG_LEVEL")) c.log_level = *v;
```

```
  if (auto v = getenv_opt("APP_REQUIRE_HTTPS")) {
    c.require_https = (*v == "1" || *v == "true" || *v == "yes");
  }
}

static void validate_or_throw(const AppConfig& c) {
  if (c.env != "development" && c.env != "staging" && c.env != "production") {
    throw std::runtime_error("APP_ENV must be development|staging|production");
  }
  if (c.http_port < 1 || c.http_port > 65535) {
    throw std::runtime_error("http_port out of range");
  }
  // Security invariant example: production should require HTTPS semantics.
  if (c.env == "production" && !c.require_https) {
    throw std::runtime_error("production requires APP_REQUIRE_HTTPS=1");
  }
}
```

Where the config file fits The above demonstrates env layering. In production, add file loading before env overrides:

```
#include <filesystem>
#include <fstream>
#include <sstream>

// Minimal INI-like parser example: key=value per line, no nesting.
// In real systems, use a well-defined format and strict parser rules.
static void apply_file_kv(AppConfig& c, const std::filesystem::path& p) {
  std::ifstream in(p);
  if (!in) throw std::runtime_error("cannot open config file");

  std::string line;
  while (std::getline(in, line)) {
```

```
    if (line.empty() || line[0] == '#') continue;
    const auto pos = line.find('=');
    if (pos == std::string::npos) continue;

    const std::string key = line.substr(0, pos);
    const std::string val = line.substr(pos + 1);

    if (key == "env") c.env = val;
    else if (key == "http_bind") c.http_bind = val;
    else if (key == "http_port") c.http_port = parse_int_in_range(val, 1, 65535,
    ↪  "http_port");
    else if (key == "log_level") c.log_level = val;
    else if (key == "require_https") c.require_https = (val == "1" || val == "true" || val
    ↪  == "yes");
  }
}
```

Final assembly

```
static AppConfig load_config(const std::filesystem::path& file_path) {
  AppConfig c;                 // defaults
  apply_file_kv(c, file_path); // file
  apply_env_overrides(c);      // env overrides
  // apply_cli_overrides(c, argc, argv); // optional CLI layer
  validate_or_throw(c);        // fail fast
  return c;
}
```

27.1.12 Operational patterns for scalability

Pattern 1: one canonical config file per environment

- `/etc/myapp/config.production.toml`

- `/etc/myapp/config.staging.toml`

Then select with:

- an environment variable or CLI flag.

Pattern 2: base file + override file

- base config holds defaults,
- override config holds environment-specific changes.

Pattern 3: immutable artifact, externalized configuration

- the binary and assets are immutable,
- all runtime variability comes from files under `/etc` and a small set of env vars.

Pattern 4: audit-friendly configuration

- config changes are made through controlled mechanisms,
- effective config summary is logged on startup (sanitized),
- restart events are traceable in systemd journal.

27.1.13 Decision guide

Prefer configuration files when:

- configuration is large or structured,
- you need readability and review,
- you want stable Linux conventions (`/etc`),
- you want explicit schema and typed validation.

Prefer environment variables when:

- you need small deployment-time overrides,
- you are integrating with systemd drop-ins or orchestration injection,
- you need quick operational toggles (with strict validation).

Never choose one blindly Use both, but keep the environment surface small and intentional.

27.1.14 Checklist: professional configuration management

- Typed config object with clear defaults.
- Explicit precedence: defaults → file → env → CLI.
- Strict validation at startup; fail fast on invalid config.
- Sanitized startup summary (no secrets).
- Secrets handled via dedicated mechanisms; avoid embedding in env/files broadly.
- systemd integration via `EnvironmentFile=` and `/etc` config paths.
- Clear operational playbooks: how to change config, how to roll back, how to verify.

27.1.15 Outcome

The most scalable configuration strategy is not "files versus environment variables"; it is a disciplined combination that assigns each mechanism an appropriate role. Configuration files provide structured, auditable, and readable configuration, while environment variables provide controlled, deployment-time overrides. With explicit precedence, strict validation, and correct secret handling, your Wt production deployments become reproducible, secure, and operationally resilient.

27.2 Managing production secrets

Managing production secrets is a foundational requirement for building secure, scalable, and operationally sound server-side applications. In professional environments, secrets are treated as first-class security assets, not as ordinary configuration values. Any failure in secret handling can directly lead to data breaches, privilege escalation, service compromise, or regulatory violations.
Production secrets include, but are not limited to:

- database usernames and passwords,
- TLS private keys and certificates,
- symmetric encryption keys,
- signing keys for authentication tokens,
- API tokens for third-party services,
- credentials for message brokers and caches,
- secret feature toggles and administrative bypass keys.

This section defines a professional, modern approach to managing secrets in Linux deployments for Wt-based applications, aligned with current industry best practices.

27.2.1 Core principles of secret management

Principle 1: Secrets are not configuration Configuration controls behavior. Secrets protect trust boundaries. Mixing them leads to accidental disclosure and weak operational controls.

Principle 2: Never embed secrets in source code Secrets must never appear in:

- source files,
- version control history,
- compiled binaries,
- test fixtures,
- CI/CD scripts.

Even private repositories and restricted build systems are not acceptable secret stores.

Principle 3: Least privilege Each service must access only the secrets it strictly requires. Broad secret access dramatically increases blast radius in the event of compromise.

Principle 4: Secrets must be rotatable Every production secret must be replaceable without requiring code changes. Long-lived static secrets are operational liabilities.

Principle 5: Secrets must never appear in logs Logging is one of the most common and dangerous leakage vectors. Secret values must be redacted by design, not by convention.

27.2.2 Acceptable secret storage mechanisms

Dedicated secret management systems

Modern production systems rely on dedicated secret stores that provide:

- encrypted storage at rest,
- authenticated and authorized access,

- fine-grained access control,
- audit trails for secret access,
- automated rotation capabilities.

In these systems:

- applications reference secrets by identifier,
- secret values are retrieved only at runtime,
- access is granted using short-lived credentials,
- every access is logged and auditable.

Filesystem-based secrets (restricted scope)

For simpler environments, secrets may be stored in files with:

- strict ownership,
- minimal permissions,
- no inclusion in backups or diagnostics,
- controlled read access by the service user only.

Plaintext files are acceptable only when:

- the filesystem is secured,
- access is tightly controlled,
- rotation and auditing procedures exist.

Environment variables: limited and controlled

Environment variables must not contain raw secret values in hardened production environments. Their acceptable uses include:

- identifying which secret to load,
- pointing to secret storage locations,
- selecting secret namespaces or environments.

Example:

```
APP_DB_SECRET_PATH=secret/data/myapp/database
SECRET_BACKEND_ADDR=secrets.internal
```

The application retrieves the secret value securely at runtime.

27.2.3 Runtime secret retrieval

Secrets must be retrieved dynamically at runtime using authenticated access. A professional retrieval flow is:

1. authenticate the service using a short-lived identity,
2. request the secret by identifier,
3. receive the secret over a secure channel,
4. store the secret only in memory,
5. discard it on shutdown or expiration.

Secrets must never be written back to disk in plaintext.

```
std::string dbPassword = secretStore.fetch("secret/data/myapp/db/password");
database.connect(dbPassword);
```

27.2.4 In-memory secret handling

Secrets stored in memory must be:

- scoped as narrowly as possible,
- cached only when necessary,
- refreshed according to defined lifetimes,
- cleared on shutdown.

Example with explicit refresh semantics:

```
struct CachedSecret {
  std::string value;
  std::chrono::steady_clock::time_point expires;
};

CachedSecret dbSecret;

std::string getDatabasePassword() {
  auto now = std::chrono::steady_clock::now();
  if (now >= dbSecret.expires) {
    dbSecret.value = secretStore.fetch("secret/data/myapp/db/password");
    dbSecret.expires = now + std::chrono::minutes(10);
  }
  return dbSecret.value;
}
```

27.2.5 Secret rotation strategy

A production-grade rotation strategy includes:

- generation of a new secret,

- updating dependent systems,
- overlapping validity period,
- application refresh or reload,
- retirement of the old secret.

Applications must tolerate secret refresh without restart whenever possible. If restart is required, it must be controlled and observable.

27.2.6 Secrets and configuration files

Configuration files may safely contain:

- secret identifiers,
- secret paths,
- access policy references.

They must never contain:

- raw passwords,
- private keys,
- token values.

Example:

```
db.host = "db.internal"
db.port = 5432
db.secret_ref = "secret/data/myapp/db"
```

27.2.7 systemd integration

When running under systemd:

- the service user must be unprivileged,
- secret access must be restricted to that user,
- environment files must contain only references, not values,
- filesystem protections must be enabled.

Example service fragment:

```
[Service]
User=myapp
Group=myapp
EnvironmentFile=-/etc/myapp/myapp.env
ProtectSystem=full
ProtectHome=true
NoNewPrivileges=true
```

27.2.8 Audit and operational controls

Professional secret management requires:

- access logging for every secret read,
- periodic review of access policies,
- detection of anomalous access patterns,
- documentation of rotation schedules,
- emergency revocation procedures.

Secrets without auditability are security liabilities.

27.2.9 Common anti-patterns

Avoid these critical mistakes:

- embedding secrets in configuration files under version control,
- passing secrets via command-line arguments,
- printing secrets during startup diagnostics,
- storing secrets in environment variables indefinitely,
- reusing secrets across environments.

27.2.10 Checklist: production-ready secret management

- Secrets stored outside source code and binaries.
- Dedicated secret storage with access control.
- Short-lived authentication to secret backends.
- In-memory caching with explicit expiration.
- Automated and testable rotation.
- No secret values in logs or diagnostics.
- Auditable access and clear ownership.
- Strict separation between configuration and secrets.

27.2.11 Outcome

Managing production secrets correctly transforms security from a fragile assumption into a controlled, auditable, and scalable system property. By externalizing secrets, enforcing least privilege, supporting rotation, and integrating securely with systemd and runtime identity mechanisms, Wt-based applications achieve production-grade security that scales with both traffic and organizational complexity.

Scalability and Maintainability

28.1 Scaling concepts

Scalability is the ability of a system to preserve correctness, responsiveness, and cost efficiency as workload grows. Maintainability is the ability to evolve that system safely over time: adding features, fixing defects, improving performance, and responding to incidents without destabilizing production.
In production-grade web systems, scaling is never just "add more servers." It is a disciplined set of architectural choices that determines:

- how load is distributed,
- where state lives,
- what becomes a bottleneck first,
- and how reliably you can operate and change the system.

For Wt applications (server-side UI, session-based interaction, optional WebSockets), scaling must account for the reality that:

- state can be maintained on the server per session,
- connections may be long-lived,

- UI event processing can be frequent and bursty,
- and resource usage often correlates with active sessions, not only request rate.

28.1.1 1) What does it mean to scale?

A scalable system is one whose throughput and capacity can grow without nonlinear failure modes. Practical definitions:

- **Capacity scaling**: handle more concurrent users, sessions, requests, or background jobs.
- **Performance scaling**: preserve latency targets (p50/p95/p99) as load rises.
- **Operational scaling**: preserve reliability as complexity and team size increase.
- **Cost scaling**: avoid wasteful overprovisioning and runaway resource consumption.

Scaling must preserve:

- **Correctness**: no broken transactions, no inconsistent state.
- **Security**: scale must not weaken authentication, authorization, or audit.
- **Predictability**: clear failure modes and bounded degradation.

28.1.2 2) Vertical scaling vs horizontal scaling

Vertical scaling (scale up)

Increase resources on one machine:

- more CPU cores,
- more RAM,

- faster storage,
- higher network bandwidth.

Pros:

- simplest operationally,
- preserves local state and in-process caches,
- minimal architectural changes.

Cons:

- has a hard ceiling,
- single-machine failure remains catastrophic,
- cost efficiency often decreases at high tiers.

Horizontal scaling (scale out)

Add more instances and distribute load:

- more Wt server processes,
- multiple hosts or containers,
- load balancer or reverse proxy distributes connections.

Pros:

- increased capacity and fault tolerance,
- rolling updates become feasible,

- more graceful handling of failures (one node down, others continue).

Cons:

- requires disciplined state management,
- requires consistent configuration and deployment automation,
- debugging becomes more complex (distributed behavior).

28.1.3 3) The state problem: stateless vs stateful scaling

Stateless scaling

Each request can be handled by any instance because:

- no important state lives only in RAM,
- state is externalized (database, cache, object store),
- session identity can be validated independently (tokens).

Stateless designs scale easily under load balancers.

Stateful scaling (typical for server-side UI frameworks)

When the application keeps per-session state in memory (widget tree, UI state, server-side interaction context), horizontal scaling must address session consistency:

- **Sticky sessions**: route a given user/session consistently to the same backend instance.
- **State externalization**: move session state out of process (difficult for rich server-side UI state; requires careful design).

Professional guidance for Wt-style statefulness:

- Use sticky sessions as the default scaling model.
- Ensure session timeouts and memory limits are strictly bounded.
- Treat “state externalization” as an advanced refactor, not a default assumption.

28.1.4 4) Scaling the connection model

Wt applications may use long-lived connections (including WebSockets). Scaling must account for:

- file descriptors (sockets),
- kernel TCP buffers,
- per-connection memory overhead,
- per-session timers and event queues.

Key concept: **concurrency is not only requests per second**. A system can be overwhelmed by many idle-but-open connections.
Operational implications:

- raise `LimitNOFILE` (file descriptor limits) appropriately,
- tune reverse-proxy timeouts to avoid premature disconnects,
- monitor active sessions and connection counts as first-class metrics.

28.1.5 5) Bottleneck taxonomy: where systems usually fail first

A professional scaling strategy begins by identifying bottlenecks. Common bottleneck classes:

CPU bottlenecks

Symptoms:

- high CPU saturation,
- rising latency with load,
- event queue backlog.

Typical causes:

- expensive per-request rendering/serialization,
- heavy templating or formatting under high frequency,
- inefficient algorithms in hot paths,
- excessive locking contention.

Memory bottlenecks

Symptoms:

- RSS grows with sessions and never comes down,
- increased paging, OOM kills,
- GC-like pauses if allocator contention becomes severe.

Typical causes:

- per-session state too large,
- caches without bounds,
- storing large uploads in memory,
- memory leaks or fragmentation.

I/O bottlenecks (disk and network)

Symptoms:

- slow response when reading/writing files,
- blocked threads waiting on disk/network,
- timeouts to upstream services.

Typical causes:

- synchronous logging to slow disks,
- synchronous file operations on request threads,
- database latency spikes,
- insufficient network bandwidth.

Database bottlenecks

Symptoms:

- rising query latency,
- connection pool exhaustion,
- lock contention or slow queries.

Typical causes:

- N+1 query patterns,
- missing indexes,
- large transactions,
- lack of caching for read-heavy workloads.

28.1.6 6) Caching as a scaling tool

Caching is a controlled trade-off: reduce repeated expensive work at the cost of freshness complexity.
Caching layers:

- **Browser cache**: cache static assets (CSS/JS/images).
- **Reverse proxy cache**: cache static responses and public resources.
- **Application cache**: cache computed results, template fragments, authorization lookups.
- **Distributed cache**: share cached data across instances (when needed).

Professional rules:

- Every cache must have a **bounded size** and **expiration policy**.
- Cache invalidation must be explicit and tested for correctness.
- Never cache secrets or private data without strict keying and access control.

28.1.7 7) Load balancing and session affinity

In horizontal scaling, a front layer distributes traffic:

- Nginx reverse proxy,
- dedicated load balancer,
- service mesh (in complex environments).

For session-based server-side UI:

- session affinity (sticky sessions) is typically required,

- the proxy must keep WebSocket upgrade and timeouts correct,
- health checks must detect dead instances quickly.

Example (conceptual) Nginx upstream with affinity:

```
upstream wt_backend {
  ip_hash;                 # simple affinity method; evaluate for your environment
  server 10.0.0.11:8080;
  server 10.0.0.12:8080;
  keepalive 64;
}
```

28.1.8 8) Decomposition: scaling the architecture, not just the process

As systems grow, responsibilities should be separated:

- **UI gateway (Wt)**: interactive sessions, forms, presentation flow.
- **Service layer**: business rules and orchestration.
- **Data layer**: database repositories and transaction boundaries.
- **Background workers**: long-running tasks (reports, exports, email).
- **Storage services**: file/object storage, CDN integration.

Key scaling concept: **move slow work out of the request path**. If a request triggers expensive work:

- queue it,
- return a job ID,
- update UI asynchronously when completed.

28.1.9 9) Concurrency model and safe scaling in Modern C++

Scaling in C++ is strongly influenced by concurrency design:

- thread-per-request models can collapse under high concurrency,
- async I/O and bounded worker pools scale more predictably,
- correct synchronization prevents hidden contention.

Professional C++ constraints:

- avoid global locks in hot paths,
- prefer immutable data and copy-on-write where appropriate,
- use connection pools for DB access,
- use bounded queues to apply backpressure.

Example: bounded queue concept (illustrative):

```
#include <condition_variable>
#include <mutex>
#include <queue>
#include <stdexcept>

template <class T>
class BoundedQueue {
  std::mutex m;
  std::condition_variable cv_push, cv_pop;
  std::queue<T> q;
  const std::size_t cap;

public:
```

```
  explicit BoundedQueue(std::size_t capacity) : cap(capacity) {}

  void push(T v) {
    std::unique_lock lk(m);
    cv_push.wait(lk, [&]{ return q.size() < cap; });
    q.push(std::move(v));
    cv_pop.notify_one();
  }

  T pop() {
    std::unique_lock lk(m);
    cv_pop.wait(lk, [&]{ return !q.empty(); });
    T v = std::move(q.front());
    q.pop();
    cv_push.notify_one();
    return v;
  }
};
```

Bounded queues are a scaling tool because they enforce backpressure rather than allowing unbounded memory growth.

28.1.10 10) Backpressure and graceful degradation

A scalable system must degrade gracefully under overload. Professional overload posture:

- reject early when capacity is exceeded,
- return clear errors for retriable failures,
- limit expensive endpoints,
- prioritize critical actions over non-critical features.

Examples of controlled degradation:

- disable optional real-time updates temporarily,
- reduce refresh frequency,
- defer report generation to background processing,
- impose per-user and per-IP rate limits.

The goal is to preserve core functionality under stress rather than fail completely.

28.1.11 11) Maintainability as a scaling multiplier

Maintainability is what allows scaling over time without accumulating fragility.
Key maintainability concepts:

- **Clear layering**: UI $\rightarrow$ controllers $\rightarrow$ services $\rightarrow$ repositories.
- **Bounded contexts**: keep domains separate; reduce coupling.
- **Stable interfaces**: evolve through versioned contracts.
- **Testability**: isolate business rules; mock external dependencies.
- **Observability**: metrics, logs, traces, and correlation IDs.
- **Automated deployment**: reproducible builds and configuration management.

A maintainable system scales engineering throughput:

- teams can change the system safely,
- performance fixes are localized,
- incidents are diagnosable and recoverable.

28.1.12 12) Operational scaling: release and failure management

Scaling requires operational maturity:

- **Health checks**: readiness and liveness.
- **Rolling updates**: replace instances without downtime.
- **Fast rollback**: revert quickly on regressions.
- **Crash recovery**: systemd restarts, rate-limit restart storms.
- **Capacity planning**: measure and forecast session growth and resource usage.

Example: systemd settings that support scaling stability:

```
[Service]
Restart=on-failure
RestartSec=5s
LimitNOFILE=65536
```

28.1.13 13) Practical scaling model for Wt deployments

A robust production model typically evolves as follows:

Stage A: Single node (vertical scaling)

- Nginx terminates TLS, proxies to Wt on localhost.
- Strict session timeouts, bounded uploads, bounded caches.
- Monitoring of sessions, CPU, memory, and latency.

Stage B: Multiple instances (horizontal scaling with affinity)

- multiple Wt instances behind Nginx/load balancer,
- session affinity enabled,
- centralized logging and consistent configuration management,
- database connection pools sized correctly across instances.

Stage C: Service decomposition

- background workers for heavy jobs,
- distributed cache for shared read-heavy data (when justified),
- storage moved to object storage or dedicated volumes with strict policy,
- performance testing and capacity planning become routine.

28.1.14 14) Scaling checklist: concepts that must be explicit

- What is your primary scaling axis: sessions, requests, or jobs?
- Where is state stored: RAM, database, cache, or hybrid?
- What is your session strategy: affinity or externalization?
- What is your bottleneck today: CPU, RAM, DB, network, locks?
- What are your overload controls: backpressure, rate limits, queue bounds?
- What are your operational guarantees: health checks, rolling updates, rollback?
- What is your observability baseline: metrics, logs, correlation IDs?

28.1.15 Outcome

Scaling concepts are the mental model that prevents fragile growth. A professional Wt deployment scales predictably when it:

- treats session state explicitly,
- controls long-lived connections and resource limits,
- removes slow work from request paths,
- uses caching responsibly with strict bounds,
- enforces backpressure and graceful degradation,
- and invests in maintainability through clean layering and observability.

This is the foundation for turning a successful application into a resilient, scalable system that remains safe to evolve under continuous production demands.

28.2 When load balancing becomes necessary

Load balancing becomes necessary when a single application instance (or a single host) can no longer meet required service objectives under real workload conditions, or when operational requirements demand fault tolerance and maintenance without downtime. In production engineering, load balancing is not merely a performance feature; it is a reliability and operability mechanism that enables:

- horizontal scaling (adding instances),
- high availability (surviving instance or host failures),
- rolling deployments (updates without full outages),

- traffic shaping and overload protection,
- and isolation of slow or failing nodes.

For Wt applications, load balancing has additional importance because Wt is often session-oriented and may use long-lived connections (including WebSockets). Therefore, the decision to introduce load balancing must consider not only request throughput but also:

- concurrent active sessions,
- socket/file descriptor limits,
- per-session memory consumption,
- long-lived connection timeouts and reconnection behavior.

This section provides a professional decision framework for recognizing when load balancing is necessary, what signals indicate it, and how to introduce it safely for Wt-based deployments.

28.2.1 1) What load balancing actually solves

Load balancing is a routing layer (software or hardware) that distributes incoming client connections across multiple backend instances. It is introduced to solve one or more of these problems:

Capacity limits

- CPU saturation under peak load.
- Memory saturation due to active session growth.
- File descriptor exhaustion due to many concurrent connections.
- Dependency pressure (database pool, caches) concentrated on one node.

Reliability limits

- A single instance crash causes total outage.
- OS reboots, kernel updates, or host maintenance require downtime.
- A memory leak forces frequent restarts that users experience as service failure.

Operational limits

- Deployments cannot be done safely without stopping the service.
- Configuration changes require downtime.
- Troubleshooting is risky because you cannot isolate traffic from a suspected bad instance.

28.2.2 2) The objective triggers: measurable signals

Load balancing should be introduced based on measurable signals, not instinct. The most defensible triggers are tied to explicit service objectives (SLOs).

Latency SLO violations under normal workload

If p95/p99 latency exceeds target during expected peak traffic, and profiling confirms the node is saturated, you need either:

- optimization, and/or
- horizontal scaling via load balancing.

Sustained CPU saturation

If a single instance runs at high CPU usage (for example, consistently near saturation during peak) and:

- throughput does not increase with additional load,
- latency rises rapidly,

then you are capacity-bound. Load balancing becomes necessary once vertical scaling no longer provides acceptable cost or headroom.

Memory pressure correlated with sessions

For Wt, memory pressure is often session-driven. Clear signals:

- RSS grows with active sessions,
- OOM events or aggressive swapping,
- frequent restarts due to memory limits,
- session churn causing spikes in allocation overhead.

When memory is the scaling axis, adding instances (and distributing sessions) becomes essential.

Connection and file descriptor exhaustion

Long-lived connections (HTTP keep-alive, WebSockets) consume resources even when idle. Signals:

- `EMFILE` errors (too many open files),
- many sockets in `ESTABLISHED` or `TIME_WAIT`,
- high connection counts correlated with incidents,
- proxy timeouts and disconnect storms.

This is a strong indicator to distribute connections across multiple instances.

Availability requirements

Even if one instance can handle the load, load balancing becomes necessary when the business requires:

- no single point of failure,
- continuous availability during host maintenance,
- rolling deploys without downtime,
- resilience to instance-level failure.

28.2.3 3) The architecture triggers: structural reasons

Certain architectural patterns make load balancing necessary early.

Session-based server-side UI

Wt applications often hold per-session server-side state. As the number of sessions grows, one process becomes a hard ceiling. Load balancing becomes necessary when:

- the session count approaches memory or CPU limits,
- per-session state complexity grows with features,
- session event traffic increases (more real-time UI updates).

Heavy background tasks sharing the same process

If the same process handles UI events and CPU-heavy tasks (exports, PDF generation, image processing), one workload can starve the other. Load balancing (or splitting workloads) becomes necessary when:

- request latency becomes correlated with background task bursts,
- thread pools become saturated,
- interactive UX degrades unpredictably.

Geographic distribution or edge requirements

If users are geographically distributed and latency matters, load balancing at regional edges may be required to:

- route traffic to nearest region,
- fail over between regions,
- absorb traffic spikes locally.

28.2.4 4) The operational triggers: when the team needs it

A mature engineering organization introduces load balancing as soon as operational needs require safe change:

Rolling updates without downtime

If you must deploy frequently, you cannot afford full stop/start windows. Load balancing enables:

- draining traffic from one instance,
- updating it,
- returning it to rotation,
- repeating across the fleet.

Canary releases

To reduce risk, you may route a small percentage of traffic to a new version. This requires a traffic distribution layer.

Isolation and incident response

When diagnosing a suspected bad instance:

- remove it from rotation,
- preserve logs and state,
- keep the service alive with remaining instances.

This ability is operationally transformative.

28.2.5 5) The Wt-specific reality: affinity and long-lived connections

When load balancing Wt, you must account for session affinity.

Why session affinity is commonly required

If the Wt server maintains UI state in memory, then requests for the same session must go to the same instance. Without affinity, users can experience:

- lost UI state,
- inconsistent session behavior,
- unexpected logouts or widget tree mismatches.

Therefore, load balancing introduces a key design decision:

- **Sticky sessions** as the standard approach for Wt-style stateful servers,
- or deep refactoring to externalize session state (advanced and expensive).

WebSockets and connection persistence

WebSockets remain bound to the chosen upstream instance for the lifetime of the connection. Load balancing must ensure:

- correct upgrade forwarding,
- long `proxy_read_timeout`,
- stability of the upstream mapping,
- safe reconnect behavior when instances roll.

28.2.6 6) Practical thresholds: when it becomes unavoidable

There is no universal numeric threshold, but there are practical patterns.

Load balancing is necessary when at least one is true:

- One instance cannot meet peak load without violating latency/error SLOs.
- You cannot maintain the server (reboot, update) without taking a full outage.
- You need safe, frequent deploys with minimal downtime.
- You are approaching resource ceilings: CPU, memory, file descriptors.
- You need resilience to node failure, not just performance.

Load balancing is urgent when:

- restart storms occur (crashes under peak),
- OOM kills happen in production,

- file descriptor limits are approached regularly,
- your only mitigation is manual restarts and prayer.

28.2.7 7) Introducing load balancing safely

A professional adoption path avoids destabilizing production.

Step 1: Make the service “load-balance ready”

- Ensure consistent configuration across instances.
- Make logging structured and include instance identifiers.
- Ensure health endpoints exist (or at least stable status checks).
- Ensure graceful shutdown works (SIGTERM drains).
- Ensure session timeouts and resource bounds are correct.

Step 2: Start with two instances

Run two Wt instances on different ports or hosts, and put Nginx (or another LB) in front.

Step 3: Enable affinity

If your Wt server is stateful, configure sticky sessions.

Step 4: Add health checks and draining

Ensure the load balancer removes unhealthy instances automatically and supports draining for deployments.

Step 5: Adopt rolling updates

Update one instance at a time while traffic continues to flow.

28.2.8 8) Minimal conceptual Nginx load balancing example

This is a conceptual upstream definition (affinity method must be chosen based on your environment):

```
upstream wt_backend {
  ip_hash;                # simple affinity; evaluate carefully for NAT-heavy clients
  server 10.0.0.11:8080;
  server 10.0.0.12:8080;
  keepalive 64;
}

server {
  listen 443 ssl http2;

  location / {
    proxy_pass http://wt_backend;

    proxy_http_version 1.1;
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Proto https;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection $connection_upgrade;

    proxy_read_timeout 3600s;
    proxy_buffering off;
  }
```

```
}
```

This example shows the mechanics. In real production, affinity and health management must be chosen deliberately.

28.2.9 9) Checklist: deciding to load balance

- Defined SLOs (latency, error rate, uptime).
- Evidence of single-instance saturation or instability.
- Resource ceilings measured: CPU, memory, file descriptors, DB pool.
- Operational requirement: rolling updates, maintenance without outage.
- Session model explicitly defined: sticky sessions vs external state.
- Reverse proxy configured for long-lived connections (WebSockets/timeouts).
- Monitoring and logging ready to handle multiple instances.

28.2.10 Outcome

Load balancing becomes necessary when growth or operational requirements make a single-instance deployment unsafe, unreliable, or unmaintainable. For Wt applications, the decision is often driven by session-driven resource pressure and long-lived connection handling. Introducing load balancing at the right time enables horizontal scaling, high availability, rolling deployments, and safer incident response, transforming a fragile server into an operable, scalable production system.

28.3 Realistic limits of Wt

Wt (WebToolkit) enables a server-side, C++-based programming model for building web applications with a widget-tree abstraction, event handling, and stateful sessions. This model can produce robust, secure systems with strong correctness guarantees and a high level of code reuse, but it also introduces realistic scaling and architecture limits that must be understood clearly in production engineering.
This section presents a professional, realistic view of Wt limits: not as a criticism, but as engineering boundaries that influence architecture, deployment, and long-term maintainability. The goal is to help you design Wt systems that remain stable, predictable, and cost-effective at scale.

28.3.1 1) The fundamental constraint: server-side state

The defining characteristic of Wt is that a significant portion of UI state may live on the server:

- widget tree state,
- event bindings,
- per-session UI context,
- server-side validation and interaction state.

Practical implication Scaling is often bounded by:

- **active session count**,
- **per-session memory footprint**,
- **per-session event rate**.

This differs from purely stateless HTTP services where a single request can be routed to any instance with minimal in-memory coupling.

28.3.2 2) Memory footprint scales with sessions

In Wt, server-side session state can be non-trivial:

- widget objects and their relationships,
- cached data for UI,
- rendering state,
- per-session timers and pending events.

Realistic limit category: memory If one session uses only a few megabytes, thousands of concurrent sessions can become a memory ceiling. Even when per-session memory is smaller, fragmentation and allocator behavior can produce growth patterns that are operationally significant.

Engineering consequences

- Session timeouts must be strict and tested.
- Abandoned sessions must be cleaned reliably.
- Large UI state must be avoided or externalized.
- Caches must be bounded and preferably shared or centralized where appropriate.

28.3.3 3) Concurrency and connection scaling limits

Wt applications often use:

- long-lived connections,
- WebSockets (or similar interactive transport),
- frequent small updates.

Realistic limit category: file descriptors and sockets Each connection consumes:

- file descriptors,
- kernel buffers,
- per-connection user-space overhead.

At scale, you can hit OS-level limits:

- open file descriptor limits (`ulimit -n` / `LimitNOFILE`),
- ephemeral port exhaustion (in certain proxy topologies),
- high `TIME_WAIT` counts,
- aggressive NAT gateways collapsing client identity for affinity.

Engineering consequences

- Raise and validate `LimitNOFILE` for the Wt service.
- Monitor active connections and socket states.
- Tune reverse-proxy timeouts for long-lived interactive sessions.
- Expect that connection count, not only RPS, defines capacity.

28.3.4 4) Sticky sessions are often required

Because session state is kept in-process, horizontal scaling typically requires session affinity:

- the same user/session must be routed to the same backend instance.

Realistic limit category: load balancing flexibility Sticky sessions:

- reduce the load balancer's ability to rebalance perfectly,
- create uneven distribution under certain client network patterns,
- complicate rolling deployments and failover.

Engineering consequences

- Load balancing must be designed around affinity.
- Health checks and draining become critical.
- Session persistence strategy must be explicit.

28.3.5 5) Failover behavior is more complex than stateless services

In stateless services, if an instance dies, new requests simply go to another instance. In Wt, if an instance dies:

- in-memory session state is lost,
- active WebSockets drop,
- users may need to reconnect and potentially re-authenticate,
- in-progress UI workflows may be interrupted.

Realistic limit category: user experience under failure This is a meaningful difference in failure semantics. The system can remain "up" but active users experience disruption.

Engineering consequences

- Design UI flows to tolerate reconnects.
- Keep critical user state in durable storage, not only widget state.
- Use transactions for important actions.
- Consider "resume" patterns for long workflows (draft save, step state).

28.3.6 6) Server-side rendering and update cost

Wt must render UI updates and serialize them to the client. Under high event rates (many users clicking, typing, updating dashboards), CPU cost can become dominant:

- diff computation and event handling,
- rendering of widget hierarchy,
- serialization and compression overhead.

Realistic limit category: CPU and event throughput A single-node can become CPU-bound even when database usage is modest.

Engineering consequences

- Avoid unnecessary re-rendering (minimize widget churn).
- Debounce frequent UI events (e.g., keystroke-level updates).
- Batch updates and reduce update frequency when possible.

- Offload heavy computations to background workers.

28.3.7 7) Long-running tasks can damage interactive responsiveness

If you run expensive tasks within the same process and thread pools used for UI:

- report generation,
- PDF creation,
- heavy image processing,
- large exports,

then interactive UX latency can spike.

Realistic limit category: coupling of interactive and batch workloads

Engineering consequences

- Move heavy tasks to asynchronous job queues.
- Use separate worker processes for CPU-heavy jobs.
- Keep the interactive Wt layer focused on session/UI responsiveness.

28.3.8 8) Observability and debugging limits at scale

Wt's server-side UI model can complicate debugging at scale because:

- a large portion of state is in memory,
- issues may be session-specific,
- reproduction requires session history and event order,
- distributed debugging becomes harder with multiple nodes and affinity.

Realistic limit category: operational complexity

Engineering consequences

- Implement request/session correlation IDs.
- Log structured events: auth, routing, critical actions.
- Capture per-session diagnostics safely (redacted).
- Add metrics for session counts, event rates, and update sizes.

28.3.9 9) Client-side ecosystem and UI constraints

Wt integrates with modern web standards, but the UI abstraction is different from mainstream client-heavy frameworks. Practical limits include:

- high-interactivity UIs may require careful tuning to avoid server load,
- large component libraries are more abundant in mainstream JS ecosystems,
- complex client-only interactions may require custom JavaScript integration.

This is not a hard technical limit, but a pragmatic constraint on developer velocity and UI breadth.

28.3.10 10) Realistic scalability envelope: what Wt can and cannot do well

Wt tends to excel when:

- business logic is complex and benefits from strong typing and C++ performance,
- security boundaries and correctness are critical,

- UI is form-driven, workflow-driven, or admin-oriented,
- server-side control of state reduces client complexity,
- latency requirements are reasonable and predictable.

Wt becomes challenging when:

- the product requires massive numbers of concurrent long-lived sessions with rich server-side state,
- the UI demands ultra-high-frequency real-time updates for large audiences,
- hard failover with seamless session continuity is mandatory,
- the organization requires rapid iteration on large client-side component ecosystems.

28.3.11 11) Practical strategies to push limits safely

You do not overcome limits by ignoring them; you overcome them by engineering around them.

Reduce per-session memory

- Keep widget trees lean.
- Avoid storing large data sets in session objects.
- Use pagination and streaming for large views.
- Store durable user state in a database, not in memory.

Bound session lifetime

- enforce idle timeouts,
- implement explicit logout,
- clear abandoned sessions reliably.

Control event rates

- debounce frequent inputs,
- batch updates,
- disable non-critical live updates under load.

Scale horizontally with affinity

- multiple Wt instances behind a load balancer,
- session affinity enabled,
- health checks and draining configured,
- fast restart and monitored crash loops.

Separate workloads

- UI servers handle sessions and event processing,
- worker processes handle batch jobs,
- storage systems handle uploads and static serving.

28.3.12 12) A concrete capacity mindset for Wt deployments

Professional capacity planning should treat these as first-class:

- maximum active sessions per instance,
- average and worst-case memory per session,
- average event rate per session,
- CPU cost per UI update,
- file descriptor usage per session,
- timeouts and reconnection behavior.

A Wt system scales safely when these numbers are measured, bounded, and monitored continuously.

28.3.13 Outcome

The realistic limits of Wt stem from its strengths: server-side state, session-oriented interaction, and a C++ control model. These strengths imply concrete scaling boundaries in memory, connection count, CPU event throughput, and operational complexity. A production-grade Wt architecture succeeds by:

- treating sessions as the primary scaling axis,
- bounding memory and event rates,
- scaling horizontally with affinity when necessary,
- separating interactive UI work from heavy background tasks,

- and investing in observability and operational discipline.

Understanding these limits early allows you to design systems that remain stable and maintainable while still benefiting from the power of Modern C++ and the Wt programming model.

Conclusion

When to Use Wt and When Not To

Comprehensive review

This final chapter consolidates the engineering lessons of the book into a single decision framework. The goal is not to "promote" Wt, nor to dismiss it, but to present a clear, professional conclusion: when Wt is a strategically correct choice for production-grade systems, when it is risky or inefficient, and how to make the decision with technical honesty. Wt is a server-side web application framework in Modern C++ that emphasizes:

- strong typing and compile-time safety,
- a widget-based UI model,
- server-side event handling and session state,
- a coherent architecture that can keep business logic close to the server.

This comes with unique strengths and unique trade-offs. The correct decision depends on workload shape, team profile, operational constraints, and the nature of the product.

1) What Wt gives you in professional terms

A unified C++ server-side engineering model

Wt allows you to implement:

- UI composition,
- validation,
- workflow orchestration,
- services and business rules,
- and data access

in a single language and toolchain.
In production, this can reduce cross-language friction and simplify correctness, especially when the backend is naturally C++ due to:

- performance requirements,
- access to existing C++ libraries,
- systems integration,
- hardware/embedded interaction,
- security-sensitive computation.

Strong correctness leverage

C++ gives:

- explicit ownership models,
- deterministic lifetimes (RAII),
- powerful static typing,
- performance predictability in critical code paths.

Wt inherits these advantages, enabling applications where correctness and performance are closely tied to business value.

Server-side control of workflows

Wt is particularly suitable for:

- form-driven systems,
- administrative consoles,
- domain workflows with many steps,
- authenticated internal applications,
- systems where server-side validation and authorization dominate.

Server-side control reduces client complexity and ensures business rules remain enforced at the boundary.

2) The trade-offs that cannot be ignored

Session state and scalability cost

The most important practical trade-off is that Wt commonly maintains server-side session state. This implies:

- per-session memory growth,
- long-lived connections,
- scaling driven by concurrent sessions rather than pure request rate,
- horizontal scaling often requiring session affinity (sticky sessions).

This is not inherently bad, but it must be treated as the primary scaling axis.

Failover semantics

If an instance fails, in-memory session state is lost. Users may experience:

- forced reconnection,
- workflow interruption,
- session recreation overhead.

Systems that require seamless continuity under failure must engineer around this explicitly (durable workflow state, savepoints, resumable flows).

UI ecosystem realities

Client-heavy ecosystems (React, Vue, etc.) have:

- a massive component ecosystem,
- fast UI iteration patterns,
- a large pool of developers,
- deep third-party integration patterns.

Wt can integrate JavaScript when needed, but the default development model differs. This impacts:

- developer hiring,
- design-system reuse,
- rapid UX experimentation.

Operational maturity requirement

Running Wt professionally requires:

- strong Linux deployment discipline,
- correct reverse proxy configuration,
- session and resource monitoring,
- service management (systemd),
- security hardening (TLS, session security, uploads).

If an organization cannot operate servers reliably, Wt will not save them; it will amplify their operational weaknesses.

3) When Wt is the right choice

Wt is an excellent choice when your system matches several of the following characteristics.

You benefit from C++ on the server

- You already have significant C++ codebase or libraries.
- You need high-performance computation integrated into web workflows.
- You require deterministic performance and low-level control.
- You need deep integration with OS, hardware, or proprietary binary components.

Your UI is workflow-driven rather than purely visual

- Business systems with forms, approvals, dashboards, admin panels.
- Systems where correctness, audit, and authorization dominate.
- Systems with complicated validation rules and transactional boundaries.

Your user base is bounded or predictable

Wt can scale well when:

- concurrency is known and capacity planning is possible,
- session memory per user can be bounded,
- session lifetimes are controlled by timeouts and real workflows.

You want strong server-side enforcement of business rules

Wt aligns well with a layered architecture:

- UI layer,
- controller layer,
- service layer,
- data layer.

This separation makes it easier to keep business rules centralized and testable.

4) When Wt is not the right choice

Wt is often the wrong choice when several of the following apply.

You need massive consumer-scale concurrency with rich real-time interaction

If you expect:

- extremely large numbers of concurrent sessions,
- high-frequency real-time UI updates for broad audiences,
- strict seamless failover for sessions,

then a server-side session model is a high-cost design. You may prefer:

- stateless backend services,
- client-heavy UI with efficient edge caching,
- specialized real-time infrastructure.

Your product is UI/UX experimentation-heavy

If you must:

- iterate UI weekly or daily,
- rely on large JS component ecosystems,
- hire large teams of frontend specialists,

then mainstream frontend frameworks often provide better velocity and staffing flexibility.

Your organization cannot operate production systems reliably

If you lack:

- monitoring and alerting discipline,

- deployment automation,
- security hardening skills,
- incident response capability,

then adopting a framework does not fix operational immaturity. Wt requires serious operations to be safe.

You require strict stateless scaling properties

If your architecture depends on:

- true statelessness,
- perfect load balancing without affinity,
- seamless horizontal scaling with minimal per-user memory,

then a server-side session UI model may be structurally mismatched.

5) Decision matrix: an engineering checklist

Use the following checklist to make an honest decision:

Workload shape

- What is your primary scaling axis: sessions, requests, or background jobs?
- Are connections long-lived (WebSockets) and how many at peak?
- Is the UI update rate high or moderate?

State strategy

- Can you accept sticky sessions?
- Can you bound per-session memory?
- Can workflows resume after reconnect?

Team and ecosystem

- Do you have C++ expertise to maintain a server safely?
- Do you need a large frontend developer pool?
- Do you depend heavily on a modern JS component ecosystem?

Operations maturity

- Can you deploy with systemd services, reverse proxy, and TLS correctly?
- Can you monitor sessions, memory, CPU, and error rates?
- Can you do rolling updates and fast rollback?

If you answer “no” to operational readiness, the priority is to mature operations before committing to any production web architecture.

6) A pragmatic hybrid model

Many successful systems use hybrid designs:

- Wt handles authenticated server-side UI workflows.
- A separate stateless API layer serves mobile apps or public endpoints.

- Background job workers handle heavy tasks.
- Object storage serves large files.
- Reverse proxy and CDN serve static assets.

Hybrid design allows Wt to be used where it shines, while avoiding it where a session-heavy server model becomes expensive.

7) Final recommendation: use Wt with clarity, not ideology

Wt is a strategic advantage when:

- C++ is a real backend requirement, not a preference,
- business workflows dominate UI requirements,
- stateful sessions are acceptable and bounded,
- you operate Linux services professionally.

Wt is a strategic liability when:

- you need consumer-scale concurrency with minimal state,
- UI iteration velocity and JS ecosystem breadth dominate,
- you require seamless session continuity under instance failures,
- operational maturity is weak.

Outcome

The correct conclusion is not that Wt is universally best or universally limited. The correct conclusion is that Wt is a specialized production tool with clear engineering boundaries. When your system benefits from Modern C++ strengths, and when you can accept and manage the realities of server-side sessions, Wt can deliver exceptionally reliable, maintainable, and high-performance applications. When your product requires stateless scaling, massive concurrency, or rapid client-heavy UI evolution, Wt becomes an unnecessary constraint, and other architectures will be objectively more efficient.
This comprehensive review is the final lesson of production engineering: choose tools based on workload reality, operational capability, and long-term maintainability—not based on fashion or ideology.

Realistic comparison with Node.js and Spring

This section provides an engineering-level comparison between **Wt (C++ server-side UI framework)**, **Node.js (JavaScript runtime + web frameworks)**, and **Spring (Java ecosystem, typically Spring Framework + Spring Boot)**. The purpose is to help you choose correctly under real-world constraints: performance, scalability, operability, team skills, and long-term maintainability.
To keep the comparison anchored to modern baselines (at the time of writing):

- Wt is actively maintained (e.g., Wt 4.12.1 released October 16, 2025).
- Node.js maintains long-term support lines (e.g., Node v24 is an LTS line; the website lists v24.13.0 as a latest LTS build).
- Spring Boot and Spring Framework continue rapid evolution (e.g., Spring Boot 4.0.1; Spring Framework 7.0.3 shown on official project pages).

These versions are cited here only to clarify that the compared ecosystems are modern and evolving; the comparison remains valid beyond exact version numbers because it is based on architectural fundamentals.

1) What each stack fundamentally is

Wt

Wt is a C++ web toolkit built around a server-side widget tree, server-side event handling, and session-driven interaction. It can run in several deployment modes (including an embedded HTTP/WebSocket server component, and connectors). The default mental model resembles a desktop GUI framework, but rendered to the browser with an event-driven server.

Node.js

Node.js is a JavaScript runtime designed for event-driven, asynchronous I/O. Web development usually uses frameworks such as Express, Fastify, Koa, NestJS, or others. The dominant model is: stateless HTTP handlers with application state externalized (database/cache) and clients often built with independent frontend frameworks.

Spring (Spring Framework + Spring Boot)

Spring is a comprehensive application framework for Java, widely used in enterprise systems. Spring Boot provides production-ready defaults and packaging for service applications. The dominant model is: layered architecture (controller/service/repository) with strong integration across data access, security, observability, messaging, and deployment patterns.

2) Architectural posture: where the complexity lives

Wt: server-side UI complexity Wt places a meaningful portion of UI logic and state on the server:

- per-session state can include widget trees and interaction context,
- server side enforces validation and workflow sequencing naturally,
- scaling is frequently tied to concurrent sessions and connection behavior.

Node.js: client + service split is common Node-based architectures commonly separate:

- a frontend (React/Vue/etc.) for rich client UI,
- stateless backend endpoints (REST/GraphQL),
- background workers for heavy jobs.

This can scale well horizontally but introduces distributed complexity (more moving parts and boundaries).

Spring: enterprise layering and integration Spring strongly supports clean layering and cross-cutting concerns:

- dependency injection and configuration profiles,
- strong patterns for transactions and persistence,
- security frameworks and policy enforcement,
- large operational ecosystem in production.

3) Concurrency and performance: what to expect realistically

Wt and C++

C++ can deliver excellent throughput and low latency, especially for CPU-heavy logic. However, overall system performance is shaped by:

- server-side UI rendering work and update frequency,
- session memory footprint and allocator behavior,
- synchronization patterns (locks/contention),
- I/O design (database, storage, network).

Wt can be extremely efficient for workflow-driven systems and services tightly coupled with C++ computation, but it is not a magic solution for unlimited interactive concurrency.

Node.js

Node is often strong at I/O-heavy workloads due to its event loop and asynchronous model, especially when:

- most time is spent waiting on network/database,
- request handlers stay short and non-blocking.

Node becomes less suitable when:

- heavy CPU work must occur inside the main process (unless offloaded to worker threads/processes),
- long synchronous operations block the event loop,
- strict tail latency (p99) is required under mixed workloads without careful engineering.

Spring

Spring runs on the JVM, which offers strong JIT optimization and mature concurrency facilities. Spring systems often perform excellently when:

- code is written with consistent architecture,
- GC tuning and memory sizing are handled correctly,
- operational practices are mature.

Spring can deliver high throughput at scale, but latency behavior depends on GC, thread pool sizing, and integration choices. The ecosystem provides many guardrails, but complexity must still be managed professionally.

4) Scalability: the honest scaling limits

Wt: sessions and affinity are central

In many Wt designs, scaling is bounded by:

- concurrent active sessions,
- per-session memory,
- long-lived connections,
- server-side event rates.

Horizontal scaling often implies:

- multiple instances behind a reverse proxy/load balancer,
- **session affinity (sticky sessions)** to keep each session on one instance.

This works well for many business systems, but it reduces the flexibility of perfect rebalancing and makes failover semantics more noticeable to users.

Node.js: horizontal scaling is natural if you stay stateless

Node services commonly scale by running many stateless replicas behind a load balancer. If state is externalized:

- any request can go to any node,
- rolling updates and failover are straightforward.

However, if you add server-side session state or WebSocket-heavy workloads, you reintroduce affinity concerns, regardless of language.

Spring: mature horizontal scaling patterns

Spring applications are frequently deployed as stateless services with externalized state and well-defined layering:

- simple load balancing and service discovery,
- mature patterns for distributed configuration,
- robust integration with messaging and async processing.

Scaling Spring is typically "well known" in the industry, but it can be heavy if you do not need the ecosystem depth.

5) Developer productivity and ecosystem realities

Wt

Strengths:

- one language for core logic and UI workflows,

- very strong integration with existing C++ libraries,
- controlled server-side enforcement of validation and business rules.

Constraints:

- smaller ecosystem compared to mainstream JavaScript frontend tooling,
- fewer ready-made UI components and design systems,
- hiring pool is narrower for “Wt specialists” than for React/Node or Spring.

Node.js

Strengths:

- huge ecosystem and developer availability,
- rapid iteration (especially with frontend + backend JS/TS),
- easy integration with modern web tooling.

Constraints:

- codebase quality depends heavily on discipline and architecture,
- large dependency surfaces can increase supply-chain risk,
- CPU-heavy tasks require explicit offloading strategy.

Spring

Strengths:

- mature enterprise ecosystem (security, data, messaging, observability),

- strong conventions for layering, testing, and deployment,
- widely understood patterns in large organizations.

Constraints:

- can be heavier than necessary for small systems,
- complexity can grow if everything is enabled without careful scope,
- requires JVM operational knowledge (memory sizing, GC awareness).

6) Security posture: what is realistic

Security is primarily a matter of engineering discipline, not language choice. Still, each stack has typical security patterns and failure modes.

Wt

- Strong advantage: business rules and validation can live on the server by default.
- Primary risks: session handling, upload handling, reverse proxy configuration, and memory-safety mistakes if unsafe C++ patterns are used.
- Professional requirement: strict input validation, safe output encoding, careful session security, and modern TLS deployment.

Node.js

- Strong advantage: mature web security middleware and community practices.
- Primary risks: dependency vulnerabilities, insecure defaults, accidental exposure of secrets in logs/env, and injection flaws from poor validation.

- Professional requirement: dependency auditing, strict request validation, security headers, correct auth and session/token handling.

Spring

- Strong advantage: robust security framework support and standard patterns for authentication/authorization.
- Primary risks: misconfiguration, overly complex security rules, and framework misuse.
- Professional requirement: minimal exposed surface, correct configuration profiles, careful policy design and testing.

7) Maintainability: long-term costs

Wt Maintainability is strong if you enforce:

- strict layering (UI/controller/service/repository),
- isolated business rules in services,
- test-first for domain logic,
- bounded session state.

Maintainability declines when:

- UI and business logic leak into each other,
- session objects become “everything bags”,
- performance fixes are done as ad-hoc hacks without architectural cleanup.

Node.js Maintainability is strong if you enforce:

- TypeScript + strict types,
- architectural boundaries (modules, services, repositories),
- controlled dependencies and clear interfaces,
- explicit async patterns and error-handling policies.

Maintainability declines when:

- business rules are scattered across route handlers,
- async flows become inconsistent,
- dependency sprawl grows without governance.

Spring Maintainability is strong if you enforce:

- clear domain layering and strict transactional boundaries,
- consistent configuration management (profiles),
- integration testing and contract testing,
- controlled module boundaries.

Maintainability declines when:

- everything becomes a global DI container,
- cross-cutting concerns are overused without discipline,
- configuration grows uncontrolled across environments.

8) Minimal code snapshots: same idea, different stacks

The following minimal examples illustrate the differences in development posture: Wt is server-side UI oriented, Node and Spring are typically endpoint/service oriented.

Wt: a tiny UI interaction

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class HelloApp final : public Wt::WApplication {
public:
  HelloApp(const Wt::WEnvironment& env) : Wt::WApplication(env) {
    setTitle("Hello Wt");

    auto* input = root()->addWidget(std::make_unique<Wt::WLineEdit>());
    auto* out   = root()->addWidget(std::make_unique<Wt::WText>(""));
    auto* btn   = root()->addWidget(std::make_unique<Wt::WPushButton>("Submit"));

    btn->clicked().connect([=] {
      out->setText("Hello, " + input->text());
    });
  }
};
```

Node.js: a minimal HTTP handler (typical stateless posture)

```
import express from "express";

const app = express();
```

```
app.use(express.json());

app.post("/hello", (req, res) => {
  const name = String(req.body?.name ?? "");
  if (!name) return res.status(400).json({ error: "name required" });
  res.json({ message: `Hello, ${name}` });
});

app.listen(3000);
```

Spring Boot: controller + service (typical layered posture)

```
@RestController
class HelloController {
  private final HelloService service;

  HelloController(HelloService service) {
    this.service = service;
  }

  @PostMapping("/hello")
  public Map<String, String> hello(@RequestBody Map<String, Object> body) {
    String name = String.valueOf(body.getOrDefault("name", ""));
    if (name.isBlank()) throw new ResponseStatusException(HttpStatus.BAD_REQUEST,
    ↪  "name required");
    return Map.of("message", service.hello(name));
  }
}

@Service
class HelloService {
  public String hello(String name) { return "Hello, " + name; }
}
```

9) Decision guidance: which one is the right tool?

Choose Wt when

- your system benefits from C++ for core computation, integration, or performance,
- UI is workflow-driven (forms, admin tools, dashboards, authenticated internal apps),
- server-side enforcement of business rules is central,
- you can accept session-driven scaling and design for it (timeouts, affinity, monitoring),
- your team can operate Linux services professionally (reverse proxy, TLS, monitoring, deployments).

Choose Node.js when

- you want fast development with a large ecosystem and hiring pool,
- your backend is primarily I/O-bound and naturally stateless,
- you rely heavily on modern frontend ecosystems and rapid UI iteration,
- you are prepared to govern dependencies and enforce architecture discipline,
- CPU-heavy work can be isolated into separate workers/services when needed.

Choose Spring when

- you need enterprise-scale integration (security, data, messaging, observability) with mature standard patterns,
- your organization already runs JVM services and has operational maturity for them,
- you want strong conventions for layered architecture and production readiness,

- your application fits the “service + integration + policy” profile common in large organizations.

10) Final reality check

A realistic comparison must end with the correct conclusion: **each stack is excellent in the environment it was designed for**.

- Wt is not a direct substitute for “React + Node” consumer UI ecosystems; it is a powerful server-side UI/workflow framework that shines when correctness, C++ integration, and server-driven workflows are central.
- Node.js is not inherently “less professional” than C++; it is an industrial standard for web systems, but requires discipline to manage dependency risk and maintain architectural quality.
- Spring is not “too heavy” by definition; it is one of the most production-proven enterprise stacks, but demands architectural scope control to avoid unnecessary complexity.

The correct decision is the one that matches your workload shape, your operational maturity, and your long-term maintainability goals.

Practical guidance for teams and projects

This section converts the entire book into practical guidance that teams can apply immediately. It focuses on organizational reality: team composition, delivery speed, quality control, operations, and long-term ownership. The objective is to make the decision about Wt (and to execute a Wt-based project) with professional clarity.
The guidance is written for real projects:

- internal enterprise applications,
- authenticated portals and dashboards,
- workflow-driven systems (approvals, case management, admin tools),
- systems integrating performance-critical C++ logic.

It also addresses the most common failure patterns observed when teams adopt server-side UI frameworks without sufficient operational discipline.

1) Team readiness: the first decision gate

Wt is not difficult because it is C++; it is demanding because it is *production web engineering* in C++. That requires:

- strong C++ fundamentals (RAII, ownership, exceptions, concurrency),
- good architecture discipline (layering, boundaries, contracts),
- Linux deployment competence (systemd, reverse proxies, TLS),
- professional security mindset (sessions, input validation, uploads),
- observability awareness (logging, monitoring, incident response).

Minimum team profile for a successful Wt project

- at least one senior C++ engineer who understands production operations,
- at least one engineer comfortable with Linux server management,
- a developer who can handle HTML/CSS/JavaScript integration when required,
- a clear code review process and CI/CD ownership.

If these roles do not exist, the project can still succeed, but it must budget time and training explicitly.

2) Project suitability: what type of product fits Wt

Projects where Wt is strategically strong

- workflow-heavy systems (forms, approvals, audit trails),
- admin panels and back-office tools,
- authenticated portals with controlled user groups,
- systems requiring strict server-side enforcement of business rules,
- products embedding complex computation (optimization, analysis, ML inference, CAD, scientific engines),
- systems where reuse of existing C++ libraries is a major advantage.

Projects where Wt is strategically risky

- consumer-scale products expecting massive concurrent sessions with rich interactivity,
- products driven by rapid frontend design-system iteration with many UI specialists,
- applications requiring seamless session continuity under node failure,
- teams with weak operations maturity and no reliable deployment pipeline.

3) Architecture rules that prevent long-term failure

A Wt project fails rarely because of Wt itself. It fails because of architecture collapse: business logic spreads, state grows uncontrolled, and deployment becomes fragile. The following rules prevent that.

Rule 1: strict layering is mandatory

Enforce a strict structure:

- **UI layer** (widgets, views, templates): presentation only.
- **Controllers** (UI orchestration): translate UI events into service calls.
- **Service layer** (business rules): transaction boundaries, validation, workflows.
- **Data layer** (repositories): SQL and persistence abstractions.

The UI must never perform:

- SQL queries,
- domain decisions,
- authorization checks beyond presentation decisions.

Rule 2: session state must be bounded and intentional

Session state is your scaling axis. Treat it as a budget:

- store only what is necessary for UI interaction,
- never store large collections or full domain aggregates in session,
- prefer identifiers and short cached summaries,
- keep durable workflow state in the database.

Rule 3: long-running work must be moved off the UI path

Anything that can take seconds (or longer) must not block interactive responsiveness:

- exports,
- report generation,
- large file processing,
- bulk imports,
- heavy computations.

Use job queues and worker processes. The UI should trigger work and poll or receive completion notifications.

4) Testing strategy: what teams must implement

A professional Wt project is test-driven at the service layer.

Test the service layer, not the widget tree

Most correctness lives in:

- business rules,
- authorization decisions,
- transactional workflows,
- validation logic.

Therefore:

- unit test services with mocked repositories,
- integration test repositories against a real database,
- add end-to-end tests only for critical UI workflows.

Define stable contracts between layers

Create explicit interfaces so services are testable:

```
struct UserRepo {
  virtual ~UserRepo() = default;
  virtual bool usernameExists(std::string_view u) = 0;
  virtual void createUser(std::string_view u, std::string_view passHash) = 0;
};

struct UserService {
  virtual ~UserService() = default;
  virtual void registerUser(std::string_view u, std::string_view password) = 0;
};
```

5) Deployment discipline: non-negotiable practices

Teams often underestimate operations. In Wt projects, operations is part of engineering.

Reverse proxy is standard

Use a reverse proxy (commonly Nginx) for:

- TLS termination,
- static asset caching,
- rate limiting and security headers,

- buffering control for uploads,
- WebSocket forwarding.

systemd is the baseline for Linux services

The service must be managed professionally:

- restart policy with backoff,
- file descriptor limits,
- sandboxing where possible,
- centralized logs.

```
[Service]
Restart=on-failure
RestartSec=5s
LimitNOFILE=65536
NoNewPrivileges=true
```

Configuration and secrets must be production-grade

- configuration is layered (defaults → file → env overrides),
- secrets are never embedded in code or normal config files,
- secret rotation is planned from the beginning.

6) Observability: the minimum monitoring set

If you cannot observe it, you cannot scale it.

Metrics you must track

- active sessions and session churn,
- memory per process and memory growth over time,
- CPU utilization and load,
- request latency percentiles (p95/p99),
- error rates and timeouts,
- connection counts and file descriptor usage,
- database pool utilization and query latency.

Logging rules

- structured logs, not free-text only,
- include correlation IDs (request/session),
- never log secrets or personal data,
- log business events that matter (login, privilege change, payment, export).

7) Scaling guidance: how teams should plan

Stage 1: single instance with discipline

This is where most teams should start:

- focus on clean architecture and testing,
- ensure session and memory bounds are correct,

- validate reverse proxy and TLS correctness,
- establish monitoring and alerting baseline.

Stage 2: scale out with session affinity

When sessions or connections exceed capacity:

- run multiple Wt instances,
- use load balancing with sticky sessions,
- implement health checks and draining for rolling updates,
- ensure database pools are sized across instances.

Stage 3: decompose workloads

As complexity grows:

- offload heavy jobs to workers,
- introduce caching where justified and bounded,
- consider separating public APIs from Wt UI.

8) Governance: keep the system maintainable over years

A production system dies from entropy unless governance exists.

Code review rules

- no business logic in UI widgets,
- no SQL outside repositories,
- mandatory validation in services,
- security checks reviewed explicitly (authz, input validation, upload limits).

Dependency governance

- minimize third-party dependencies,
- pin and update libraries intentionally,
- record upgrade procedures,
- regularly audit security issues.

Documentation and runbooks

- deployment steps and rollback procedures,
- incident response playbooks,
- operational dashboards and alert thresholds,
- architecture decision records for major choices.

9) Practical decision summary for teams

A team should choose Wt when

- C++ is a core strength and business requirement,

- the UI is workflow-driven and server enforcement matters,
- the user base is bounded or predictable,
- the team can operate Linux services professionally,
- long-term maintainability and correctness matter more than frontend fashion.

A team should avoid Wt when

- ultra-high concurrency with minimal per-user state is required,
- rapid UI experimentation and large JS ecosystems dominate,
- seamless session continuity under node failures is mandatory,
- operations and deployment maturity are not present.

Outcome

Wt can be an outstanding production platform when teams treat it as a professional engineering stack: strict layering, bounded session state, tested business logic, disciplined deployment, strong security posture, and continuous observability. The practical lesson is simple: Wt rewards disciplined teams and punishes architectural shortcuts. Choose it for the right product shape, run it with operational maturity, and it will deliver maintainable, production-grade systems that fully exploit Modern C++ strengths.

Appendices

Appendix A Common Errors and Troubleshooting

A.1 Troubleshooting Mindset for Production Wt

When a Wt application misbehaves in production, the fastest path to root cause is to treat the system as a pipeline:

- **Client** (browser, HTTP cache, cookies, WebSocket availability)
- **Edge** (CDN / WAF, TLS termination, reverse proxy, request/idle timeouts)
- **App Server** (Wt process, configuration, session management, thread pool)
- **Dependencies** (database, file system permissions, external HTTP calls, mail/SMS gateways)
- **OS** (ports, DNS, firewall, SELinux/AppArmor, cgroups/systemd limits)

A production-safe workflow:

1. **Reproduce with minimal variables**: same URL, same headers, same proxy route, same user role.

2. **Collect authoritative evidence**: server logs, reverse proxy access/error logs, application stderr/stdout, DB logs.
3. **Isolate layers**: test direct-to-app (bypass proxy) versus through proxy; test HTTP versus WebSocket; test with fresh session.
4. **Fix the narrowest cause**: one change at a time; roll forward with a rollback plan.

A.2 Build and Link Errors

A.2.1 CMake cannot find Wt

Symptoms

- `Could not find a package configuration file provided by "Wt"`
- `Wt_INCLUDE_DIR not found` (legacy find modules)

Common causes

- Wt not installed to a standard prefix, or `CMAKE_PREFIX_PATH` not set.
- Mixing a system Wt with a custom-built Wt.
- Using a Wt build tree instead of the installed export targets.

Fix

```
# Prefer imported targets if your Wt installation exports them
set(CMAKE_PREFIX_PATH "/opt/wt;/usr/local" ${CMAKE_PREFIX_PATH})

find_package(Wt REQUIRED COMPONENTS Wt HTTP)

add_executable(app main.cpp)
target_link_libraries(app PRIVATE Wt::Wt Wt::HTTP)
```

If you build and install Wt yourself, confirm you installed it (not only built it), and confirm the installation contains `WtConfig.cmake` and exported targets.

A.2.2 Undefined references or ODR-like link failures

Symptoms

- `undefined reference to Wt::...`
- Multiple definitions across translation units when using unity builds or aggressive LTO.

Common causes

- Linking only `Wt` but not the needed component (`HTTP`, `Dbo`, `DboSqlite3`, `DboPostgres`, etc.).
- ABI mismatch: different compiler, different standard library, different build flags (e.g., `_GLIBCXX_USE_CXX11_ABI` on older toolchains).
- Mixing Debug and Release libraries on Windows/MSVC.

Fix

- Link all required component targets explicitly.
- Keep compiler + STL + build type consistent across Wt and your application.
- Avoid mixing static and shared builds unless you control all dependencies carefully.

A.2.3 OpenSSL / TLS build problems

Symptoms

- `cannot find -lssl` / `-lcrypto`

- Runtime: `error while loading shared libraries: libssl.so...`

Fix checklist

- Install OpenSSL development headers and runtime libs.
- Ensure runtime loader paths are correct (`ldconfig` on Linux).
- Prefer system OpenSSL on production unless you have a controlled vendor build.

A.3 Startup and Port Binding Issues

A.3.1 Server fails to start or cannot bind

Symptoms

- `bind(): Address already in use`
- `Permission denied` when binding to ports below 1024
- Service starts and immediately exits

Common causes

- Another process already listens on the configured port.
- The process lacks privileges for privileged ports.
- Misconfigured `-docroot`, missing config file, or missing runtime directories.

Fix

- Use a non-privileged port (e.g., 8080/8443) behind a reverse proxy.
- On Linux, use capabilities instead of root if you must bind to 80/443:

```
sudo setcap 'cap_net_bind_service=+ep' /usr/local/bin/my_wt_app
```

A.3.2 Working directory and relative path failures

Symptoms

- `cannot open resource`, missing CSS/JS, missing templates/images
- Works when run from a terminal but fails under systemd

Cause System managers often set a different working directory. Relative paths resolve differently.

Fix

- Use absolute paths in configuration.
- For systemd, set `WorkingDirectory=` and explicit `Environment=` values.

```
# /etc/systemd/system/my_wt_app.service
[Service]
WorkingDirectory=/srv/myapp
Environment=WT_CONFIG_XML=/etc/myapp/wt_config.xml
ExecStart=/usr/local/bin/my_wt_app --config=${WT_CONFIG_XML}
Restart=on-failure
```

A.4 Reverse Proxy and Deployment Problems

A.4.1 Wrong client IP / scheme behind proxy

Symptoms

- Application thinks requests are `http` while the public site is `https`
- Generated absolute URLs are wrong
- Logs show proxy IP instead of client IP

Common causes

- Missing or untrusted `X-Forwarded-For` / `X-Forwarded-Proto`.
- Proxy terminates TLS and forwards plain HTTP to the app, but the app isn't configured to trust forwarded headers.

Fix

- Configure your reverse proxy to send standard forwarded headers.
- Configure the Wt server/application to operate correctly behind a proxy (deployment mode and proxy settings).

A.4.2 WebSockets fail (upgrade fails, 400/502, or forced long-polling)

Symptoms

- UI feels sluggish, frequent full refreshes, or frequent reconnects
- Browser console shows failed `WebSocket` handshake or `Upgrade` errors
- Reverse proxy errors: `upstream prematurely closed connection`, `bad gateway`

Common causes

- Proxy not forwarding `Upgrade` and `Connection` headers.
- Proxy read timeout too small for persistent connections (WebSocket or long polling).
- HTTP/2 or buffering settings conflicting with upgrade/push endpoints.

Example Nginx reverse proxy configuration (core ideas)

```
location / {
  proxy_pass http://127.0.0.1:8080;

  # Preserve host and scheme info
  proxy_set_header Host $host;
  proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
  proxy_set_header X-Forwarded-Proto $scheme;

  # WebSocket upgrade
  proxy_http_version 1.1;
  proxy_set_header Upgrade $http_upgrade;
  proxy_set_header Connection "upgrade";

  # Long-running connections need generous timeouts
  proxy_read_timeout  3600s;
  proxy_send_timeout  3600s;

  # Avoid buffering for interactive streaming
  proxy_buffering off;
}
```

A.4.3 Sessions time out unexpectedly behind a proxy

Symptoms

- Users are logged out while still active
- Session expires exactly at a proxy timeout boundary
- Intermittent behavior: works locally, fails behind load balancer/CDN

Common causes

- Proxy request timeout smaller than Wt server push / long-poll settings.
- Idle connection policies in the edge layer kill long polling.
- Sticky sessions not enabled when multiple app instances are used and sessions are not shared.

Fix

- Align proxy timeouts with Wt push/keep-alive behavior (proxy must allow long polling / WebSocket to remain open).
- If you scale horizontally, either:
  - enable **sticky sessions** on the load balancer, or
  - use a **shared session strategy** compatible with your deployment design.

A.5 TLS / HTTPS Troubleshooting

A.5.1 TLS handshake fails or app will not start with SSL enabled

Symptoms

- App exits at startup when SSL is configured
- Reverse proxy can connect, but direct HTTPS fails

Common causes

- Wrong certificate/key file path or permissions.
- Missing intermediate chain when terminating TLS in the app server (if not using a proxy).

- Using a certificate/key format not supported by the configured OpenSSL build.

Fix checklist

- Confirm the process user can read the key file (strict permissions).
- Validate certificate + key pairing and chain completeness.
- Prefer TLS termination at a hardened reverse proxy for most production deployments.

A.6 Authentication, Cookies, and CSRF-Like Failures

A.6.1 Login loops or "it works locally but fails in production"

Symptoms

- Users log in, then immediately appear logged out
- Session cookie never persists

Common causes

- Cookie flags mismatch (`Secure` cookies over HTTP, wrong `SameSite` behavior).
- Mixed scheme (`http` vs `https`) due to reverse proxy misreporting scheme.
- Clock skew or inconsistent session secrets across replicas.

Fix

- Ensure public site is consistently HTTPS and forwarded scheme is correct.
- Ensure all replicas share the same security/session configuration when scaling out.
- Verify browser cookie policy changes if embedding in iframes or cross-site flows.

A.7 Wt::Dbo and Database Errors

A.7.1 Connection failures and driver misconfiguration

Symptoms

- Exceptions on startup: cannot connect, authentication fails, unknown database
- Works on developer machine, fails on server

Common causes

- Missing runtime client libraries on the server.
- Wrong connection parameters, DNS, or firewall.
- Using a DB server version with strict defaults that break legacy assumptions.

Fix checklist

- Verify DB connectivity from the app host (network, credentials, TLS if used).
- Ensure the correct backend library is deployed with the application.
- Log connection parameters safely (never log passwords).

A.7.2 Schema mismatch or migration problems

Symptoms

- Exceptions during transaction start or first query
- Runtime errors after deploying a new version

Common causes

- Entity mapping changed without applying a migration.
- Concurrent deployments with different schemas.

Fix

- Apply forward-only migrations before switching traffic.
- For zero-downtime deployments: keep backward compatibility during the migration window.

A.7.3 Concurrency conflicts in transactions

Symptoms

- Conflicts under load: optimistic locking detects concurrent updates

Fix strategy

- Treat transaction failures as normal: implement bounded retries for transient conflicts.
- Keep transactions short: do not perform network calls inside DB transactions.

```
for (int attempt = 0; attempt < 3; ++attempt) {
  try {
    Wt::Dbo::Transaction trx(session);
    // ... load, modify, persist ...
    trx.commit();
    break;
  } catch (const Wt::Dbo::StaleObjectException&) {
    if (attempt == 2) throw;
    // retry with fresh read
  }
}
```

A.8 Performance and Stability Issues

A.8.1 High CPU usage with many connected clients

Symptoms

- CPU spikes as concurrency increases
- Latency rises; UI updates become delayed

Common causes

- Excessive server-side UI updates (too many signals, too frequent timers).
- Rendering large widget trees frequently instead of incremental updates.
- Overuse of synchronous blocking work on request threads.

Fix

- Debounce UI updates; batch changes; avoid per-keystroke server round-trips for heavy work.
- Move blocking work to background threads; push results back safely.
- Profile at production-like load (sampling profiler + request timing logs).

A.8.2 Memory growth, leaks, or unbounded caches

Symptoms

- RSS grows over time, GC-like behavior absent (native process)
- OOM kills under load

Common causes

- Session-scoped objects never released because sessions never expire (keep-alive policies).
- Capturing `shared_ptr` in long-lived callbacks/timers unintentionally.
- Unbounded in-memory caches (images, DB query results, templates).

Fix checklist

- Audit ownership: prefer clear lifetimes (session vs application global).
- Use bounded caches with eviction.
- Ensure sessions expire and are cleaned up appropriately.

A.9 Static Resources, Paths, and MIME Problems

A.9.1 Missing CSS/JS/images in production

Symptoms

- Page renders without styling or widgets misbehave
- 404 for resources or wrong content-type

Common causes

- Incorrect `docroot` or resource deployment path.
- Reverse proxy path rewriting not aligned with app base path.
- MIME types not served correctly by the proxy or by the app server.

Fix

- Confirm resource paths are correct when accessed through the proxy.

- Verify the reverse proxy does not strip or rewrite essential prefixes.
- Ensure correct MIME mapping for common assets (CSS, JS, fonts, SVG).

A.10 Logging and Diagnostics You Should Always Enable

A.10.1 Minimum production diagnostics

At minimum, ensure you can answer these questions from logs alone:

- Which request failed, for which user/session, at what time?
- Did it fail in the proxy layer or in the app?
- Was the failure transient (timeout) or deterministic (config/schema)?

Recommended logging practices:

- Log **request id** and propagate it through your application and DB calls.
- Log **session id** (or a safe surrogate) and user id (if authenticated).
- Log timings for: routing, DB, external calls, rendering.
- Avoid logging secrets; redact tokens and passwords.

A.11 Production Readiness Checklist (Quick Troubleshooting Map)

A.11.1 If the app is down

- Port bind failure or process crash: check service logs and crash dumps.
- Proxy 502/503: check upstream health, firewall rules, and app listen address.
- TLS errors: confirm certificates, key permissions, and proxy TLS termination.

A.11.2 If the app is slow

- Measure: proxy timings vs upstream timings.
- Check DB latency and pool saturation.
- Check thread pool starvation caused by blocking work.
- Reduce chatty UI updates; batch server-side changes.

A.11.3 If sessions disconnect or expire

- Align proxy timeouts with Wt push/keep-alive behavior.
- Confirm WebSocket upgrade forwarding and idle timeouts.
- If multiple replicas: ensure sticky sessions or shared session strategy.

A.12 Common "Gotchas" That Waste the Most Time

- **Relative paths under systemd**: always validate `docroot` and config paths.
- **Proxy timeouts**: default reverse proxy settings often kill long polling or WebSockets.
- **Mixed scheme**: HTTPS outside, HTTP inside, but the app generates wrong absolute URLs unless proxy headers are correct.
- **DB schema drift**: deploy migrations before switching traffic.
- **Unbounded session state**: keep session memory small and lifetimes explicit.

Appendix B Ready-to-Use CMake Templates

B.1 Goals of These Templates

This appendix provides production-oriented **copy/paste** CMake templates for Wt-based C++ web applications. Each template is designed to be:

- **Modern**: target-based CMake, explicit features/standards, strict warnings.
- **Portable**: Linux (GCC/Clang), Windows (MSVC), macOS (Clang).
- **Production-ready**: sane defaults for build types, LTO (optional), sanitizers (optional), install rules (optional).
- **Wt-aligned**: clean `find_package(Wt ...)` usage and explicit component linking.

B.2 Assumptions and Conventions

- CMake version: use a **recent** CMake (the templates specify a modern minimum).
- C++ standard: C++20 by default (adjust to C++23 if your toolchain supports it).
- Layout: `src/` contains code, optional `resources/` contains static resources/docroot.
- Wt package: these templates assume Wt is installed and provides a usable CMake package.

B.3 Template 1 Minimal Single-Executable Wt Application

Use when: you want the smallest clean project that compiles and links with Wt quickly.

Directory Layout

```
myapp/
  CMakeLists.txt
  src/
    main.cpp
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.24)

project(MyWtApp
  VERSION 0.1.0
  LANGUAGES CXX
)

# ----- Global policy: prefer target-based configuration
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

# Default build type for single-config generators (Ninja/Make)
if(NOT CMAKE_CONFIGURATION_TYPES AND NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
endif()

# Wt provides multiple components; HTTP is typical for Wt::WServer usage.
find_package(Wt REQUIRED COMPONENTS Wt HTTP)

add_executable(myapp
```

```
  src/main.cpp
)

target_link_libraries(myapp
  PRIVATE
    Wt::Wt
    Wt::HTTP
)

# A good baseline for compile warnings (tune for your coding standard)
if(MSVC)
  target_compile_options(myapp PRIVATE /W4 /permissive-)
else()
  target_compile_options(myapp PRIVATE -Wall -Wextra -Wpedantic)
endif()
```

src/main.cpp (minimal skeleton)

```
#include <Wt/WApplication.h>
#include <Wt/WEnvironment.h>
#include <Wt/WText.h>

class App : public Wt::WApplication {
public:
  App(const Wt::WEnvironment& env) : Wt::WApplication(env) {
    setTitle("My Wt App");
    root()->addWidget(std::make_unique<Wt::WText>("Hello from Wt"));
  }
};
```

```
std::unique_ptr<Wt::WApplication> createApp(const Wt::WEnvironment& env) {
  return std::make_unique<App>(env);
}

int main(int argc, char** argv) {
  return Wt::WRun(argc, argv, &createApp);
}
```

B.4 Template 2 Production Baseline (Warnings, Options, Hardening Switches)

Use when: you want a solid default baseline with toggles for sanitizers, LTO, and strict compilation.

Directory Layout

```
myapp/
  CMakeLists.txt
  cmake/
    Tooling.cmake
  src/
    main.cpp
```

`CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.24)

project(MyWtProdApp
  VERSION 1.0.0
  LANGUAGES CXX
```

```
)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

if(NOT CMAKE_CONFIGURATION_TYPES AND NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
endif()

# Options (safe defaults)
option(MYAPP_ENABLE_WERROR     "Treat warnings as errors" OFF)
option(MYAPP_ENABLE_SANITIZERS "Enable ASan/UBSan (non-MSVC, Debug/RelWithDebInfo)"
↪  OFF)
option(MYAPP_ENABLE_LTO        "Enable LTO/IPO if supported" OFF)

list(APPEND CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}/cmake")
include(Tooling)

find_package(Wt REQUIRED COMPONENTS Wt HTTP)

add_executable(myapp
  src/main.cpp
)

target_link_libraries(myapp PRIVATE Wt::Wt Wt::HTTP)

myapp_set_common_warnings(myapp)
myapp_apply_werror(myapp MYAPP_ENABLE_WERROR)
```

```
myapp_apply_sanitizers(myapp MYAPP_ENABLE_SANITIZERS)
myapp_apply_lto(myapp MYAPP_ENABLE_LTO)
```

cmake/Tooling.cmake

```
include(CheckIPOSupported)

function(myapp_set_common_warnings target)
  if(MSVC)
    target_compile_options(${target} PRIVATE
      /W4
      /permissive-
      /Zc:__cplusplus
    )
  else()
    target_compile_options(${target} PRIVATE
      -Wall -Wextra -Wpedantic
      -Wconversion -Wsign-conversion
      -Wshadow -Wformat=2
    )
  endif()
endfunction()

function(myapp_apply_werror target opt)
  if(${opt})
    if(MSVC)
      target_compile_options(${target} PRIVATE /WX)
    else()
      target_compile_options(${target} PRIVATE -Werror)
    endif()
```

```
  endif()
endfunction()

function(myapp_apply_sanitizers target opt)
  if(NOT ${opt})
    return()
  endif()

  if(MSVC)
    message(STATUS
    ↪  "Sanitizers: MSVC settings vary; prefer clang-cl or configure separately.")
    return()
  endif()

  # Typically use sanitizers in Debug or RelWithDebInfo
  if(CMAKE_BUILD_TYPE MATCHES "Debug|RelWithDebInfo")
    target_compile_options(${target} PRIVATE -fsanitize=address,undefined
    ↪  -fno-omit-frame-pointer)
    target_link_options(${target} PRIVATE -fsanitize=address,undefined)
  else()
    message(STATUS
    ↪  "Sanitizers requested, but build type is ${CMAKE_BUILD_TYPE}. Prefer Debug/RelWith
  endif()
endfunction()

function(myapp_apply_lto target opt)
  if(NOT ${opt})
    return()
  endif()
```

```
  check_ipo_supported(RESULT ipo_supported OUTPUT ipo_error)
  if(ipo_supported)
    set_property(TARGET ${target} PROPERTY INTERPROCEDURAL_OPTIMIZATION TRUE)
  else()
    message(STATUS "LTO/IPO not supported: ${ipo_error}")
  endif()
endfunction()
```

B.5 Template 3 Wt + Database Layer (Wt::Dbo) Skeleton

Use when: your Wt application includes a database layer using Wt::Dbo and a backend (SQLite/PostgreSQL/MySQL).

Notes

- Wt::Dbo is split into components; your installation may provide targets for the core and for specific backends.
- Prefer linking imported targets. If your environment names differ, keep the structure but adjust component/target names accordingly.

`CMakeLists.txt` (illustrative)

```
cmake_minimum_required(VERSION 3.24)

project(MyWtDboApp LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

```
if(NOT CMAKE_CONFIGURATION_TYPES AND NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
endif()

# Request core Wt + HTTP + Dbo (backend components vary by installation)
find_package(Wt REQUIRED COMPONENTS Wt HTTP Dbo)

add_executable(myapp
  src/main.cpp
  src/model.cpp
  src/model.hpp
)

target_link_libraries(myapp PRIVATE
  Wt::Wt
  Wt::HTTP
  Wt::Dbo
)

if(MSVC)
  target_compile_options(myapp PRIVATE /W4 /permissive-)
else()
  target_compile_options(myapp PRIVATE -Wall -Wextra -Wpedantic)
endif()
```

B.6 Template 4 Install Rules and Runtime Layout

Use when: you want **make install** / packaging-friendly output with a predictable runtime layout.

Suggested Runtime Layout

```
/opt/myapp/
  bin/myapp
  etc/myapp/wt_config.xml
  share/myapp/docroot/...
  share/myapp/templates/...
  var/myapp/ (writable runtime state, optional)
```

`CMakeLists.txt` (install section)

```
include(GNUInstallDirs)

# Install executable
install(TARGETS myapp
  RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)

# Install configuration and resources (example paths; adapt as needed)
install(DIRECTORY resources/docroot/
  DESTINATION ${CMAKE_INSTALL_DATADIR}/myapp/docroot
)

install(FILES config/wt_config.xml
  DESTINATION ${CMAKE_INSTALL_SYSCONFDIR}/myapp
)
```

B.7 Template 5 Multi-Target Layout (Core Library + Server Executable)

Use when: you want a clean separation between application logic and the web entry point. This is strongly recommended for testability and maintainability.

Directory Layout

```
myapp/
  CMakeLists.txt
  src/
    app/
      app.cpp
      app.hpp
    server/
      main.cpp
```

`CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.24)

project(MyWtLayeredApp LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

if(NOT CMAKE_CONFIGURATION_TYPES AND NOT CMAKE_BUILD_TYPE)
  set(CMAKE_BUILD_TYPE Release CACHE STRING "Build type" FORCE)
endif()

find_package(Wt REQUIRED COMPONENTS Wt HTTP)
```

```
add_library(myapp_core STATIC
  src/app/app.cpp
)

target_include_directories(myapp_core PUBLIC
  ${CMAKE_CURRENT_SOURCE_DIR}/src
)

target_link_libraries(myapp_core PUBLIC
  Wt::Wt
)

add_executable(myapp_server
  src/server/main.cpp
)

target_link_libraries(myapp_server PRIVATE
  myapp_core
  Wt::HTTP
)

if(MSVC)
  target_compile_options(myapp_core PRIVATE /W4 /permissive-)
  target_compile_options(myapp_server PRIVATE /W4 /permissive-)
else()
  target_compile_options(myapp_core PRIVATE -Wall -Wextra -Wpedantic)
  target_compile_options(myapp_server PRIVATE -Wall -Wextra -Wpedantic)
endif()
```

B.8 Template 6 Toolchain-Friendly Presets (CMakePresets.json)

Use when: you want reproducible builds across machines (CI, dev, staging) without retyping flags.

CMakePresets.json

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 24,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "linux-release",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/linux-release",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release",
        "MYAPP_ENABLE_LTO": "ON",
        "MYAPP_ENABLE_WERROR": "OFF",
        "MYAPP_ENABLE_SANITIZERS": "OFF"
      }
    },
    {
      "name": "linux-dev",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/build/linux-dev",
```

```
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "RelWithDebInfo",
        "MYAPP_ENABLE_LTO": "OFF",
        "MYAPP_ENABLE_WERROR": "OFF",
        "MYAPP_ENABLE_SANITIZERS": "ON"
      }
    }
  ],
  "buildPresets": [
    { "name": "build-linux-release", "configurePreset": "linux-release" },
    { "name": "build-linux-dev",     "configurePreset": "linux-dev" }
  ]
}
```

B.9 Template 7 Cross-Platform Notes and Fixes

B.9.1 Windows (MSVC) notes

- Prefer a consistent toolchain: build Wt and your app with the same MSVC version and runtime configuration.
- Be strict about Debug vs Release library matching.
- If consuming Wt through a package manager, ensure the triplet/config matches your build.

B.9.2 Linux notes

- Prefer system packages for Wt and its dependency stack where possible.
- When using a custom install prefix, export it via `CMAKE_PREFIX_PATH`.

- If you enable sanitizers, test with production-like concurrency (sanitizers add overhead).

B.9.3 macOS notes

- Ensure OpenSSL and other dependencies are consistently resolved (system vs package manager).
- Treat deployment paths carefully; prefer install rules for predictable runtime layout.

B.10 Troubleshooting These Templates

B.10.1 `find_package(Wt)` fails

- Confirm Wt is installed and provides CMake package configuration files.
- Provide the prefix explicitly:

```
cmake -S . -B build -DCMAKE_PREFIX_PATH="/opt/wt;/usr/local"
```

B.10.2 Linker errors

- Ensure you link all required Wt components (e.g., `HTTP`, `Dbo`).
- Ensure compiler/ABI/build-type consistency between Wt and your app.

B.10.3 Runtime cannot load shared libraries

- Ensure the runtime loader can locate shared libraries (Linux: `ldconfig` and proper library paths).
- Prefer install layouts and service definitions that set working directory and environment explicitly.

B.11 Practical “Copy and Build” Commands

```
# Configure
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release

# Build
cmake --build build

# Run (example; your Wt app arguments may differ)
./build/myapp --docroot . --http-address 0.0.0.0 --http-port 8080
```

Appendix C systemd and Nginx Configuration Templates

C.1 Overview

This appendix presents robust, ready-to-use configuration templates for deploying Wt-based applications using `systemd` and `nginx`. The focus is on reliability, security, and maintainability. The templates assume a modern Linux server environment such as Debian/Ubuntu or RHEL/CentOS and that the Wt application and its resources are installed under a predictable layout.

C.2 Runtime Layout Conventions

The recommended runtime layout for a production deployment is:

```
/opt/mywtapp/
  bin/mywtapp
  etc/wt_config.xml
  share/docroot/
  share/templates/
```

```
  var/log/mywtapp/
  var/run/mywtapp/
```

This isolates binaries, configs, static assets, and runtime state.

C.3 systemd Unit File Template

The following systemd unit file template runs your Wt application as a managed service with restart policies and environment configuration.

```
# /etc/systemd/system/mywtapp.service
[Unit]
Description=Wt Application Service
After=network.target
Wants=network-online.target

[Service]
Type=simple
User=www-data
Group=www-data
WorkingDirectory=/opt/mywtapp
Environment=WT_CONFIG=/opt/mywtapp/etc/wt_config.xml
ExecStart=/opt/mywtapp/bin/mywtapp --config=${WT_CONFIG}
Restart=on-failure
RestartSec=5s
# Limit runtime resources (adjust based on your capacity planning)
LimitNOFILE=65536
TimeoutStartSec=30
TimeoutStopSec=30
# Create /opt/mywtapp/var/run/mywtapp as a runtime directory
```

```
RuntimeDirectory=mywtapp
RuntimeDirectoryMode=0755

[Install]
WantedBy=multi-user.target
```

C.4 systemd Timer for Log Rotation (Optional)

If your application produces logs without logrotate integration, you can add a simple daily timer.

```
# /etc/systemd/system/mywtapp-logrotate.service
[Unit]
Description=Rotate Wt App Logs

[Service]
Type=oneshot
ExecStart=/usr/sbin/logrotate /etc/logrotate.d/mywtapp

# /etc/systemd/system/mywtapp-logrotate.timer
[Unit]
Description=Daily Log Rotation for Wt App Logs

[Timer]
OnCalendar=daily
Persistent=true

[Install]
WantedBy=timers.target
```

C.5 Logging Considerations

With systemd, use `journalctl -u mywtapp.service` to inspect stdout/stderr. Ensure your Wt app emits structured logs to stdout (JSON or key-value) so they integrate well with journald.

C.6 Nginx Reverse Proxy Template

This nginx configuration assumes your Wt application listens on localhost port `8080`. It handles TLS termination, proxy headers, WebSocket upgrades, and caching of static assets.

C.6.1 Core Proxy Configuration

```
# /etc/nginx/sites-available/mywtapp.conf
server {
    listen 80;
    server_name example.com www.example.com;
    # Redirect all HTTP to HTTPS
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl http2;
    server_name example.com www.example.com;

    # TLS certificate and key (letsencrypt or other CA)
    ssl_certificate /etc/ssl/certs/example.com.crt;
    ssl_certificate_key /etc/ssl/private/example.com.key;
    ssl_protocols TLSv1.3 TLSv1.2;
    ssl_prefer_server_ciphers on;
    ssl_session_cache shared:SSL:10m;
```

```
    ssl_session_timeout 1h;

    # Strong defaults for security
    add_header Strict-Transport-Security "max-age=31536000; includeSubdomains;
    ↪  preload" always;
    add_header X-Frame-Options DENY;
    add_header X-Content-Type-Options nosniff;
    add_header Referrer-Policy "no-referrer-when-downgrade";

    # Proxy configuration for Wt application
    location / {
        proxy_pass http://127.0.0.1:8080;

        # Preserve host and client info
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        # WebSocket upgrade support
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        # Timeouts optimized for long polling / WebSockets
        proxy_read_timeout 3600s;
        proxy_send_timeout 3600s;
        proxy_connect_timeout 30s;
```

```
        # Disable buffering for interactive streaming
        proxy_buffering off;
    }

    # Static files can be cached aggressively (optional)
    location ~* \.(?:css|js|jpg|jpeg|png|gif|ico|svg|woff2?)$ {
        expires 30d;
        access_log off;
        add_header Cache-Control "public, must-revalidate";
    }
}
```

C.7 Common nginx Tuning Parameters

- **worker_processes auto**: lets nginx scale to available CPU cores.
- **worker_connections 10240**: supports many simultaneous incoming connections.
- **client_max_body_size 10M (or higher)**: adjust if your app expects larger uploads.
- Logging: configure separate access/error logs for this site for easier analysis.

C.8 Example nginx Global Config Additions (/etc/nginx/nginx.conf)

```
user www-data;
worker_processes auto;
worker_rlimit_nofile 65536;

events {
    worker_connections 10240;
    multi_accept on;
```

```
}

http {
    sendfile on;
    tcp_nopush on;
    tcp_nodelay on;
    keepalive_timeout 65;
    types_hash_max_size 2048;

    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    access_log  /var/log/nginx/access.log;
    error_log   /var/log/nginx/error.log;

    # gzip compression (optional)
    gzip on;
    gzip_types text/plain text/css application/json application/javascript text/xml
    ↪  application/xml application/xml+rss text/javascript;
    gzip_min_length 1024;
    gzip_proxied any;
}
```

C.9 Deployment Checklist

- Verify DNS entries point to your servers public IP.
- Generate and install TLS certificates (Lets Encrypt recommended).
- Place the Wt binary and config under the chosen runtime layout.

- Enable and start the systemd service:

```
sudo systemctl daemon-reload
sudo systemctl enable mywtapp.service
sudo systemctl start mywtapp.service
```

- Enable the nginx site configuration and restart nginx:

```
sudo ln -s /etc/nginx/sites-available/mywtapp.conf /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

C.10 Verification and Health Checks

- Confirm the Wt application is listening on the expected port:

```
sudo ss -tlnp | grep mywtapp
```

- Confirm nginx serves TLS correctly:

```
curl -I https://example.com
```

C.11 Common Pitfalls

- **Missing proxy headers**: Without `X-Forwarded-Proto` and `Host`, Wt may generate incorrect URLs.
- **WebSocket upgrade failure**: Ensure `proxy_set_header Upgrade` and `Connection "upgrade"` are set.
- **Timeout mismatch**: Long polling/WebSockets can break if nginx or the upstream proxy prematurely closes idle connections.
- **Permissions for systemd**: Ensure the selected user (e.g., `www-data`) owns or can access required directories.

Appendix D Widget Cheat Sheet

D.1 Purpose and How to Use This Cheat Sheet

This appendix is a compact, production-focused reference for the most commonly used Wt widgets and patterns. It is written as a **practical lookup tool**:

- **What it is**: the widgets role in a UI.
- **When to use it**: common real-world scenarios.
- **Key API patterns**: the few calls that matter most day-to-day.
- **Production notes**: performance, security, maintainability pitfalls.

D.2 Core Concepts You Must Internalize (Quick Reference)

Ownership and lifetime

- **Widgets are owned**: typically by a parent container (e.g., `root()`), via `std::unique_ptr`.
- Creating widgets is commonly done with `addWidget(std::make_unique<...>())`.
- A widgets lifetime matches its parent container; removing/destroying the parent destroys children.

Signals and slots

- Most interaction is driven by signals (e.g., `clicked()`, `changed()`, `entered()`).
- You typically connect a lambda capturing the needed state safely.
- Avoid capturing raw pointers to widgets that may be deleted before the callback runs.

Layout vs CSS

- Use **layout managers** for structural placement (rows/columns/containers).
- Use **CSS classes and themes** for styling, spacing, and responsive behavior.
- Prefer CSS for typography and spacing; prefer layouts for structure and reflow logic.

Server-side UI model

- Wt UI lives on the server; the browser receives DOM updates.
- Keep UI event handlers fast; offload heavy work to background tasks.
- Minimize chatty updates in high-frequency events (typing, mouse move).

D.3 Minimal Interaction Skeleton (Baseline Pattern)

Use this as a reusable mental template for most UI interactions.

```
#include <Wt/WApplication.h>
#include <Wt/WContainerWidget.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>

class App : public Wt::WApplication {
public:
  App(const Wt::WEnvironment& env) : Wt::WApplication(env) {
    setTitle("Widget Cheat Sheet");

    auto container = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
    auto label = container->addWidget(std::make_unique<Wt::WText>("Ready"));
```

```
    auto btn = container->addWidget(std::make_unique<Wt::WPushButton>("Click"));

    btn->clicked().connect([label] {
      label->setText("Clicked");
    });
  }
};

std::unique_ptr<Wt::WApplication> createApp(const Wt::WEnvironment& env) {
  return std::make_unique<App>(env);
}

int main(int argc, char** argv) {
  return Wt::WRun(argc, argv, &createApp);
}
```

D.4 Containers and Layout

D.4.1 `WContainerWidget`

What it is: a generic container that holds child widgets and can be styled with CSS.
When to use:

- Group UI blocks (cards, panels, page sections).
- Apply CSS classes to a group.
- Provide a parent for layouts.

Key patterns:

```
auto box = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
```

```
box->addStyleClass("card");
box->setPadding(10);
```

D.4.2 `WTemplate`

What it is: server-side HTML template with placeholders and bound widgets.
When to use:

- Build consistent page skeletons (header/body/footer).
- Reuse markup and keep layout readable.

Key patterns:

```
#include <Wt/WTemplate.h>
#include <Wt/WText.h>

auto t =
↪  root()->addWidget(std::make_unique<Wt::WTemplate>(Wt::WString::tr("page")));
t->bindWidget("title", std::make_unique<Wt::WText>("Dashboard"));
```

D.4.3 Layouts: `WVBoxLayout`, `WHBoxLayout`, `WGridLayout`

What they are: layout managers that control child placement.
When to use:

- VBox: forms and vertical stacks.
- HBox: toolbars and horizontal action rows.
- Grid: label/value forms, dashboards, responsive-ish structures.

Key patterns:

```
#include <Wt/WVBoxLayout.h>
#include <Wt/WLineEdit.h>
#include <Wt/WPushButton.h>

auto panel = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
auto layout = panel->setLayout(std::make_unique<Wt::WVBoxLayout>());

auto name = layout->addWidget(std::make_unique<Wt::WLineEdit>());
name->setPlaceholderText("Name");

auto save = layout->addWidget(std::make_unique<Wt::WPushButton>("Save"));
layout->setSpacing(8);
```

D.5 Text, Links, and Content

D.5.1 `WText`

What it is: text output widget.
When to use:

- Labels, headings, dynamic messages.
- Status lines (connected, saving, error).

Production note: by default, prefer safe text rendering; only use raw HTML modes when you fully control the content.

```
#include <Wt/WText.h>

auto msg = root()->addWidget(std::make_unique<Wt::WText>("Hello"));
msg->addStyleClass("muted");
```

D.5.2 `WAnchor`

What it is: link widget.
When to use:

- Navigation links.
- External documentation links (careful with target and rel attributes if applicable).

```
#include <Wt/WAnchor.h>
#include <Wt/WLink.h>

auto a =
↪  root()->addWidget(std::make_unique<Wt::WAnchor>(Wt::WLink("https://example.com"),
↪  "Open"));
```

D.6 Inputs and Form Controls

D.6.1 `WLineEdit`

What it is: single-line text input.
When to use:

- Username, search, short values.

Key signals:

- `changed()` for value change semantics.
- `enterPressed()` for submission patterns.

```
#include <Wt/WLineEdit.h>
```

```
auto q = root()->addWidget(std::make_unique<Wt::WLineEdit>());
q->setPlaceholderText("Search...");
q->enterPressed().connect([q] {
  auto text = q->text().toUTF8();
  (void)text;
});
```

D.6.2 `WTextArea`

What it is: multi-line text input.
When to use:

- Comments, descriptions, notes.

```
#include <Wt/WTextArea.h>

auto ta = root()->addWidget(std::make_unique<Wt::WTextArea>());
ta->setRows(6);
ta->setPlaceholderText("Write your notes...");
```

D.6.3 `WPushButton`

What it is: button widget.
When to use:

- Actions (Save, Delete, Submit).

Production note: disable the button during long operations to prevent double-submits.

```
#include <Wt/WPushButton.h>

auto save = root()->addWidget(std::make_unique<Wt::WPushButton>("Save"));
```

```
save->clicked().connect([save] {
  save->setDisabled(true);
  // perform action...
  save->setDisabled(false);
});
```

D.6.4 `WCheckBox` and `WRadioButton`

What they are: boolean option and exclusive choice controls.

```
#include <Wt/WCheckBox.h>

auto cb = root()->addWidget(std::make_unique<Wt::WCheckBox>("Remember me"));
cb->changed().connect([cb] {
  bool on = cb->isChecked();
  (void)on;
});
```

D.6.5 `WComboBox`

What it is: drop-down selection.
When to use:

- Enumerations, roles, statuses, page size controls.

```
#include <Wt/WComboBox.h>

auto combo = root()->addWidget(std::make_unique<Wt::WComboBox>());
combo->addItem("Low");
combo->addItem("Medium");
combo->addItem("High");
```

```
combo->changed().connect([combo] {
  int idx = combo->currentIndex();
  (void)idx;
});
```

D.6.6 Numeric inputs: `WSpinBox`, `WDoubleSpinBox`

When to use:

- Bounded integers/floats with step control.

```
#include <Wt/WSpinBox.h>

auto n = root()->addWidget(std::make_unique<Wt::WSpinBox>());
n->setRange(0, 100);
n->setValue(10);
```

D.7 Navigation and Multi-Page UI

D.7.1 `WStackedWidget`

What it is: a container that shows one child at a time.
When to use:

- Simple multi-page navigation without changing the URL.
- Wizard flows and settings tabs.

```
#include <Wt/WStackedWidget.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
```

```
auto stack = root()->addWidget(std::make_unique<Wt::WStackedWidget>());
auto p1 = stack->addWidget(std::make_unique<Wt::WText>("Page 1"));
auto p2 = stack->addWidget(std::make_unique<Wt::WText>("Page 2"));
(void)p1; (void)p2;

auto next = root()->addWidget(std::make_unique<Wt::WPushButton>("Next"));
next->clicked().connect([stack] {
  int i = stack->currentIndex();
  stack->setCurrentIndex((i + 1) % stack->count());
});
```

D.7.2 Menus: `WMenu`

What it is: binds a menu to a `WStackedWidget`.

```
#include <Wt/WMenu.h>
#include <Wt/WStackedWidget.h>
#include <Wt/WText.h>

auto contents = root()->addWidget(std::make_unique<Wt::WStackedWidget>());
auto menu = root()->addWidget(std::make_unique<Wt::WMenu>(contents));

menu->addItem("Home", std::make_unique<Wt::WText>("Home content"));
menu->addItem("Settings", std::make_unique<Wt::WText>("Settings content"));
```

D.8 Tables and Data Views

D.8.1 `WTable`

What it is: simple HTML table widget.
When to use:

- Small tables with manual cell control.

```
#include <Wt/WTable.h>
#include <Wt/WText.h>

auto t = root()->addWidget(std::make_unique<Wt::WTable>());
t->elementAt(0, 0)->addWidget(std::make_unique<Wt::WText>("Name"));
t->elementAt(0, 1)->addWidget(std::make_unique<Wt::WText>("Value"));
```

D.8.2 MVC widgets: `WTableView`, `WTreeView`

What they are: views backed by models (scalable datasets, sorting, filtering).
When to use:

- Large tables requiring lazy loading, sorting, filtering, pagination.
- Hierarchical structures with expansions (tree).

Production note: prefer model/view for large datasets to avoid server-side widget explosion.

D.9 Dialogs and Notifications

D.9.1 `WDialog`

What it is: modal dialog container.
When to use:

- Confirmations, editing forms, wizards.

```
#include <Wt/WDialog.h>
#include <Wt/WPushButton.h>
#include <Wt/WText.h>
```

```
auto d = std::make_unique<Wt::WDialog>("Confirm");
d->contents()->addWidget(std::make_unique<Wt::WText>("Proceed?"));
auto ok = d->footer()->addWidget(std::make_unique<Wt::WPushButton>("OK"));
ok->clicked().connect([dlg = d.get()] {
  dlg->accept();
});
d->show();
```

D.9.2 `WMessageBox` (quick prompts)

When to use:

- Quick alerts and confirmations.

D.10 File Uploads and Downloads

D.10.1 `WFileUpload`

What it is: upload control.
Production notes:

- Validate size, type, and content server-side.
- Store uploads outside the web root; never trust filename.

```
#include <Wt/WFileUpload.h>
#include <Wt/WPushButton.h>

auto up = root()->addWidget(std::make_unique<Wt::WFileUpload>());
auto send = root()->addWidget(std::make_unique<Wt::WPushButton>("Upload"));

send->clicked().connect([up] {
```

```
  up->upload();
});
up->uploaded().connect([up] {
  auto tmp = up->spoolFileName();
  (void)tmp;
});
```

D.10.2 `WResource` / `WStreamResource` (downloads)

When to use:

- Secure server-side file generation (CSV, PDF exports).

D.11 Media and Graphics

D.11.1 `WImage`

When to use:

- Static images; use `WLink` or resources for dynamic images.

D.11.2 Charts

Wt provides charting capabilities for common needs (line, bar, pie) when backed by models. For high-frequency real-time dashboards, prefer batching updates and limiting redraws.

D.12 Styling and Themes (Practical Cheat List)

- **CSS classes**: `addStyleClass()`, `removeStyleClass()` for state changes.
- **Inline style**: use sparingly; prefer classes for consistency.

- **Responsive design**: use CSS media queries and container layouts; keep server-side layout simple.

```
auto w = root()->addWidget(std::make_unique<Wt::WContainerWidget>());
w->addStyleClass("card shadow p-3");
```

D.13 Security-Oriented Widget Usage Notes

D.13.1 Output encoding

- Treat all user-provided strings as untrusted.
- Prefer text APIs that escape by default; avoid injecting raw HTML.

D.13.2 Forms and state

- Validate inputs server-side even if you validate in the UI.
- Disable double-submission; add idempotency where appropriate.

D.13.3 File uploads

- Enforce limits (size/type), scan if required, store outside docroot.
- Never reflect user filenames directly into paths.

D.14 Performance-Oriented Widget Usage Notes

- Avoid huge widget trees for large datasets; use model/view widgets.
- Debounce high-frequency events; batch updates.
- Move slow tasks off request/UI thread; update UI when results arrive.

D.15 Quick Index (What to Choose When)

- **Text output**: `WText`
- **Single-line input**: `WLineEdit`
- **Multi-line input**: `WTextArea`
- **Action button**: `WPushButton`
- **Boolean option**: `WCheckBox`
- **Drop-down**: `WComboBox`
- **Numeric bounded**: `WSpinBox` / `WDoubleSpinBox`
- **Tabs/pages without URL**: `WStackedWidget` + `WMenu`
- **Small manual table**: `WTable`
- **Large data table**: `WTableView` (model/view)
- **Dialog**: `WDialog` or `WMessageBox`
- **Uploads**: `WFileUpload`
- **Downloads**: `WResource` / `WStreamResource`

Appendix E Roadmap for Large-Scale Wt Projects

E.1 What "Large-Scale" Means in a Wt Context

A large-scale Wt system is not defined by the number of source files; it is defined by **operational pressure**:

- **Concurrency**: thousands of simultaneous sessions, long-lived connections, or heavy peak bursts.
- **Complex UI**: multiple modules, role-based pages, dashboards, rich forms, background updates.
- **Data volume**: large tables, historical logs, analytics, event streams, file storage.
- **Organization scale**: multiple developers, CI/CD, compliance/security requirements, staged environments.
- **Reliability targets**: SLOs/SLAs, zero-downtime deployments, predictable latency.

E.2 Guiding Principles (Non-Negotiables)

1) Architecture before features Large Wt projects succeed when the UI layer is treated as **a thin orchestrator** and business logic remains in a well-tested core.

2) Explicit lifetimes and ownership Because Wt is server-side C++, lifetime mistakes become production outages (leaks, stale callbacks, concurrency hazards). You must enforce clear ownership boundaries.

3) Treat every external dependency as unreliable Database, filesystem, mail, HTTP APIs, queues: all must be wrapped with timeouts, retries (bounded), and resilience patterns.

4) Observability is a feature If you cannot measure it, you cannot operate it. Logging, metrics, tracing, and health endpoints are first-class.

E.3 Roadmap Phases (From Prototype to Enterprise)

This roadmap is intentionally staged: each phase produces a stable foundation that can survive the next growth step.

E.3.1 Phase 0 Foundation and Project Skeleton (Days to Weeks)

Objective: create a repo that scales with people and features.

Deliverables

- Repository layout: `core/`, `web/`, `infra/`, `tests/`, `docs/`.
- Build system: target-based CMake, presets, options for sanitizers/LTO.
- Coding standard and tooling: formatting, static analysis, warnings policy.
- Basic Wt app skeleton with routing/navigation pattern.
- Dependency policy: pin versions, reproducible builds, artifact strategy.

Large-scale rule

- If it is not testable outside Wt, it does not belong in Wt.

E.3.2 Phase 1 Domain Core and Service Boundaries (Weeks)

Objective: implement business logic as a library, independent of UI.

Deliverables

- **Domain model** (entities, value objects, invariants).
- **Application services** (use cases) with explicit inputs/outputs.

- **Ports and adapters** style: repositories, email sender, file store, HTTP clients.
- **Error model**: typed errors, mapping to UI messages safely.
- **Unit tests**: domain and service tests run in CI quickly.

Core interface example (UI-independent)

```
#pragma once
#include <string>
#include <expected>

struct CreateUserRequest {
  std::string email;
  std::string displayName;
};

enum class CreateUserError {
  InvalidEmail,
  EmailAlreadyUsed,
  StorageFailure
};

struct UserId { std::string value; };

class IUserService {
public:
  virtual ~IUserService() = default;
  virtual std::expected<UserId, CreateUserError>
  createUser(const CreateUserRequest& req) = 0;
};
```

E.3.3 Phase 2 Production UI Layer Pattern (Weeks to Months)

Objective: scale the UI safely without turning it into business logic.

UI structure

- **Pages**: each page is a composable widget module (dashboard, admin, reports).
- **Presenters/ViewModels**: optional layer to translate service DTOs to UI state.
- **Components**: reusable widgets (forms, tables, dialogs, notifications).
- **Navigation**: consistent pattern for routing and access control.

Deliverables

- Role-based access control at a single choke point.
- UI conventions: error banners, empty states, loading states, pagination, search patterns.
- Centralized input validation:
  - client-side hints for UX
  - server-side validation for truth
- UI performance policy: avoid large widget trees; use model/view for large datasets.

Practical large-scale Wt UI rule

- No page may execute heavy work inside a signal handler without moving it to a background workflow.

E.3.4 Phase 3 Data Layer Hardening (Months)

Objective: prevent the database from becoming the system bottleneck or failure source.

Deliverables

- Connection pool policy, timeouts, and retry strategy (bounded).
- Migrations: forward-only, automated, applied before traffic shift.
- Transaction discipline:
  - short transactions
  - no external calls inside transactions
  - clear isolation expectations
- Read-model patterns for heavy dashboards (avoid expensive joins in hot paths).
- Integration tests against a real DB in CI (containerized if possible).

Retry policy example for transient conflicts

```
template <class Fn>
auto retry_transient(int maxAttempts, Fn&& fn) {
  for (int i = 1; i <= maxAttempts; ++i) {
    try {
      return fn();
    } catch (...) {
      if (i == maxAttempts) throw;
    }
  }
  throw; // unreachable
}
```

E.3.5 Phase 4 Observability and Operations (Months, then Continuous)

Objective: achieve predictable operations under real load.

Deliverables

- **Structured logs**: request id, session id, user id (safe), latency, error category.
- **Metrics**: request rate, error rate, p50/p95/p99 latency, memory, CPU, thread pool saturation.
- **Tracing (optional but powerful)**: correlate UI action → service call → DB queries.
- **Health endpoints**: readiness (deps OK) vs liveness (process alive).
- **Alerting**: SLO-based, not noise-based.

Logging discipline

- Never log secrets (tokens, passwords).
- Log errors as typed categories, not raw text only.
- Make logs machine-parseable for incident response.

E.3.6 Phase 5 Scaling Strategy (Horizontal, Vertical, and Architectural)

Objective: scale without redesigning everything at crisis time.

Scale options

- **Vertical scaling**: simplest first step (more CPU/RAM).
- **Horizontal scaling**: multiple app instances behind a load balancer.
- **Asynchronous backends**: queues/workers for slow tasks (exports, emails, imports).
- **Service decomposition**: only when organizational and operational maturity exists.

Critical Wt scaling decision If you run multiple instances:

- Ensure session strategy is correct:
  - sticky sessions, or
  - a shared session design aligned with your security model.

When to introduce background workers

- Any task that takes longer than a user-interaction budget (typically seconds).
- Any task that spikes CPU/memory or calls external APIs.
- Any task that must be retried independently of the user session lifetime.

E.4 Security Roadmap (Build It In, Not After)

E.4.1 Phase-by-phase security

- **Early**: secure defaults, HTTPS, strict cookie policy, safe output encoding.
- **Mid**: centralized authN/authZ, audit logging, rate limits on sensitive endpoints.
- **Late**: secret management, key rotation, dependency scanning, threat modeling.

E.4.2 Wt-specific security rules

- Never trust client-side validation; always validate server-side.
- Treat any user-provided text rendered as HTML as a security boundary.
- For file uploads: validate content type and size; store outside docroot; generate server-side filenames.
- Use explicit timeouts for external calls; limit payload sizes.

E.5 Performance Roadmap (Avoid the Classic Wt Pitfalls)

E.5.1 UI update strategy

- Debounce high-frequency events (typing).
- Batch updates: set multiple widget states, then flush once.
- Avoid per-row widget creation for huge tables; use model/view widgets.

E.5.2 Thread pool discipline

- Do not block request/UI threads with slow I/O.
- Create a dedicated worker pool for background work.
- Protect shared resources with clear ownership and minimal locking scope.

E.5.3 Caching strategy

- Cache stable resources (templates, static data) with clear invalidation rules.
- Prefer bounded caches; avoid unbounded `std::unordered_map` caches in process memory.
- Distinguish:
  - in-process cache (fast, per-instance)
  - distributed cache (shared, extra operational cost)

E.6 Team and Process Roadmap (Scaling People)

E.6.1 Standards that prevent large-project collapse

- Enforce code review with checklists: security, performance, correctness, tests.
- CI gates: build, unit tests, integration tests, static analysis.
- Branching policy and release cadence with staged environments.
- Architecture decision records (ADRs) for major decisions.

E.6.2 Documentation artifacts that matter

- System architecture diagram and deployment diagram.
- Runbooks: incident response, restart procedures, rollback steps.
- Onboarding guide: local dev setup, debugging, profiling, test strategy.

E.7 Suggested Milestones (A Practical Timeline)

The following milestones are realistic for building a production-grade large-scale Wt application:

Milestone 1 "Shippable MVP"

- Minimal features, but full production skeleton: CI, logging, health checks, deployment automation.

Milestone 2 "Operational Confidence"

- Metrics dashboards, alerting, load testing baseline, capacity planning notes.

Milestone 3 "Scale Readiness"

- Horizontal scaling tested, session strategy validated, background workers introduced.

Milestone 4 "Enterprise Hardening"

- Audit logging, secrets management, security testing pipeline, formal runbooks.

E.8 Load Testing and Capacity Planning Checklist

- Define realistic user scenarios: login, navigation, search, form submission, exports.
- Measure:
  - p50/p95/p99 latency
  - memory growth per session
  - CPU per active session
  - DB query latency distribution
- Validate failure modes: DB down, slow external API, disk full, certificate expired.
- Verify recovery: restart service, rollback deployment, clear queue backlog.

E.9 Final Large-Scale Rules (Print This)

1. Keep Wt UI thin: UI orchestrates, core computes.
2. Make lifetimes explicit: no ambiguous ownership in callbacks.
3. Keep transactions short and deterministic.
4. Never block UI/request threads with slow work.

5. Observability is mandatory, not optional.

6. Design for scale early: session strategy, deployment strategy, caching policy.

7. Treat security as a default: safe rendering, strict validation, secure storage.

References

How to Read This References Chapter

This chapter lists **trusted, primary and widely accepted secondary references** that support the engineering decisions, APIs, security practices, and deployment guidance used throughout this book.

- **Primary sources** are official standards, specifications, vendor documentation, and authoritative project documentation.
- **Secondary sources** are well-established professional references (books, reputable technical knowledge bases) used for interpretation, patterns, and best practices.
- All items are presented **without URLs** to keep this appendix paste-ready and stable for print.

Wt Toolkit Documentation

Official Wt Documentation (Primary)

- **Wt Reference Manual (API Documentation)**
  Emweb / Wt Project.

The authoritative API reference generated from the Wt source tree. Use for exact class names, overloads, signals, ownership rules, and configuration options.

- **Introduction to Wt (Tutorial)**
  Emweb / Wt Project.
  Practical tutorial introducing the server-side widget model, event handling, and application bootstrapping.

- **Introduction to Wt::Dbo**
  Emweb / Wt Project.
  Official guidance for the ORM layer: sessions, transactions, mappings, and typical database patterns.

- **Introduction to Wt::Auth**
  Emweb / Wt Project.
  Official guidance for authentication primitives, password handling patterns, and login flows.

- **Wt Configuration and Deployment Notes**
  Emweb / Wt Project documentation set.
  Use for server configuration options (HTTP/HTTPS settings, docroot, resources), reverse-proxy compatibility expectations, and operational concerns.

Wt Source and Examples (Primary, Engineering-Grade)

- **Wt Source Repository (Official)**
  Emweb / Wt Project.
  The definitive behavior reference for edge cases, undocumented interactions, performance characteristics, and real patterns used by maintainers.

- **Wt Examples and Demo Applications**

Emweb / Wt Project.
Real-world usage patterns for widgets, resources, layouts, charts, authentication, and Dbo-backed applications.

ISO C++ Standards

The Standard Itself (Primary)

- **ISO/IEC 14882 — Programming Languages — C++**
  International Organization for Standardization (ISO) / International Electrotechnical Commission (IEC).
  The normative specification of the C++ language and library as implemented by conforming compilers and standard libraries.

WG21 and Standardization Materials (Primary)

- **WG21 Working Drafts and Committee Papers (N/P papers)**
  ISO C++ Committee (WG21).
  Design rationale, feature wording, library evolution proposals, and defect reports that explain *why* and *how* modern C++ features were specified.

- **WG21 Core and Library Issue Trackers**
  ISO C++ Committee (WG21).
  Defect reports, clarifications, and post-publication corrections; essential for precise interpretation of tricky rules and library corner cases.

Compiler and Standard Library Conformance (Primary Vendor Docs)

- **GCC C++ Standards Support and Library Documentation**

GNU Project.
Implementation details, supported feature sets, ABI notes, and practical constraints for production builds.

- **Clang/LLVM C++ Standards Support and libc++ Documentation**
  LLVM Project.
  Language/library support levels, sanitizer tooling behavior, and deployment-facing build guidance.

- **MSVC C++ Conformance and STL Documentation**
  Microsoft.
  Conformance notes, build flags, standard library behavior, and Windows-specific ABI/runtime constraints.

C++ Language and Architecture References

Authoritative Language and Library References (Secondary, Widely Trusted)

- **cppreference.com — C++ Reference**
  Community-maintained, widely used technical reference with strong alignment to the standard and active maintenance.
  Use for quick lookup of library APIs, feature tables, and practical examples (verify corner cases against ISO wording when needed).

- **C++ Core Guidelines**
  Herb Sutter, Bjarne Stroustrup, and contributors.
  Practical best practices for correctness, safety, resource management, concurrency, and API design.

- **The C++ Programming Language**
  Bjarne Stroustrup.
  Foundational language description and design perspective; useful for conceptual grounding and idioms.

- **Effective Modern C++**
  Scott Meyers.
  Modern idioms and pitfalls related to move semantics, type deduction, smart pointers, and concurrency basics.

Performance, Concurrency, and Low-Level Engineering (Secondary)

- **C++ Concurrency in Action**
  Anthony Williams.
  Practical concurrency patterns, memory model awareness, and correct synchronization strategies for production systems.

- **Computer Systems: A Programmer's Perspective**
  Bryant & O'Hallaron.
  Systems fundamentals: processes, virtual memory, linking, performance, and the OS boundary relevant to production C++ servers.

- **Linkers and Loaders**
  John R. Levine.
  Deep linking/loading fundamentals supporting reliable deployment and debugging of native services.

Enterprise Software Architecture Sources

Core Architecture and Design References (Secondary, Canonical)

- **Domain-Driven Design: Tackling Complexity in the Heart of Software**
  Eric Evans.
  Domain modeling, bounded contexts, ubiquitous language, and long-lived architecture strategy.

- **Patterns of Enterprise Application Architecture**
  Martin Fowler.
  Transaction scripts vs domain model, repository patterns, unit of work, identity map, and practical enterprise layering.

- **Clean Architecture**
  Robert C. Martin.
  Dependency inversion across boundaries, use-case-driven application services, and testability-first layering.

- **Refactoring (and Refactoring Catalog)**
  Martin Fowler and contributors.
  Safe change strategies essential for large Wt projects with continuous evolution.

Modern Distributed Systems and Data-Intensive Design (Secondary)

- **Designing Data-Intensive Applications**
  Martin Kleppmann.
  Consistency, replication, transactions, messaging, and operational realities for production-grade web systems.

- **Release It!**
  Michael T. Nygard.
  Stability patterns, timeouts, bulkheads, circuit breakers, and operations-aware design.

Architecture Communication and Documentation (Secondary)

- **The C4 Model for Software Architecture**
  Simon Brown.
  Practical diagrams for system context, containers, components, and code-level communication.

- **Software Architecture Documentation Practices**
  Widely adopted documentation patterns for ADRs, runbooks, and operational playbooks.

Web Security and Deployment Guidelines

Web Application Security (Primary/Community Standards)

- **OWASP Top 10 (Latest Edition)**
  Open Worldwide Application Security Project (OWASP).
  Risk-oriented baseline for web application security priorities.

- **OWASP Application Security Verification Standard (ASVS)**
  OWASP.
  A structured, testable security requirements baseline suitable for enterprise-grade assurance.

- **OWASP Cheat Sheet Series**
  OWASP.

Practical, implementation-level guidance (authentication, session management, password storage, input validation, logging, and secure headers).

Internet Protocol and Web Standards (Primary)

- **HTTP Semantics and Related HTTP RFCs**
  IETF RFC Editor / IETF HTTP Working Group.
  Normative behavior for HTTP, caching semantics, headers, status codes, and intermediaries (reverse proxies).

- **TLS 1.3 Specification**
  IETF RFC Editor.
  Normative security and handshake behavior for modern HTTPS deployments.

- **Recommendations for Secure Use of TLS and DTLS**
  IETF RFC Editor.
  Best-practice guidance for safe protocol configuration and avoiding legacy pitfalls.

Browser and Web Platform Security Baselines (Primary/Authoritative)

- **MDN Web Docs: HTTP, Security Headers, Cookies, and Web Security**
  Mozilla.
  Practical, developer-facing explanations aligned with standards and real browser behavior (cookies, SameSite, CSP, CORS, HSTS, etc.).

Deployment and Operations (Primary)

- **systemd Documentation and Manual Pages**
  systemd project.

Authoritative behavior for service units, security hardening directives, restart policies, watchdogs, resource limits, and journald logging.

- **Nginx Official Documentation**
  Nginx project.
  Reverse proxy configuration, TLS termination, buffering/timeouts, WebSocket upgrade forwarding, and production tuning.

- **Linux Distribution Documentation (RHEL/Debian/Ubuntu families)**
  Vendor documentation sets.
  System administration behaviors that affect production services: users/groups, permissions, firewalling, SELinux/AppArmor, logging, packaging, and updates.

Security Governance and Enterprise Controls (Primary)

- **NIST Security and Identity Guidance (Selected SP Publications)**
  National Institute of Standards and Technology (NIST).
  Security control baselines, authentication assurance, and operational guidance that influence enterprise deployments and compliance programs.

Practical Reading Order (Recommended)

1. Wt Reference Manual + Wt Intro (build correct mental model and API usage)
2. ISO/IEC 14882 + WG21 notes (resolve language and library edge cases correctly)
3. OWASP Top 10 + OWASP ASVS (define security requirements and priorities)
4. HTTP/TLS RFCs + Nginx/systemd docs (make deployments correct under proxies and TLS)

5. Enterprise architecture books (keep the codebase sustainable at scale)