https://simplifycpp.org

DRAFT

Rust Backend Engineering

Prepared by Ayman Alheraki

Rust Backend Engineering

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

Prepared by Ayman Alheraki

February 2026

Contents

Author’s Introduction

A Personal Engineering Perspective

I come to Rust from more than three decades of professional experience with C and C++. For over 30 years, I worked deeply in systems programming, low-level design, performance-critical applications, and architecture-heavy software projects.
C++ shaped the way I think about software:

- Ownership and resource management.
- Performance as a design requirement.
- Explicit control over memory and concurrency.
- Strong type systems as engineering tools.

When I first approached Rust, I did so with the skepticism of someone who has seen many languages rise and fade. However, I quickly discovered something different.

Why Rust Attracted Me

Rust is not simply another systems language. It is a language that enforces discipline without sacrificing performance.

Despite being nearly as demanding as C++ in terms of conceptual depth, I found Rust:

- More structured.
- More consistent in design.
- More explicit in error handling.
- More disciplined in concurrency safety.

The borrow checker, often perceived as difficult, felt to me like a well-designed static analysis tool integrated directly into the compiler. It does not restrict creativity; it enforces correctness. For someone with long experience in C++, Rust feels familiar in power, yet cleaner in structure and more coherent in philosophy.

Why This Booklet Focuses on Web Backend

Many modern backend systems are built using popular stacks such as NodeJS and JavaScript-based frameworks. These tools are productive and widely adopted, but they are not always suitable for every class of project.
There exist backend systems that require:

- High throughput under heavy request loads.
- Strict memory control.
- Deterministic performance characteristics.
- Strong safety guarantees.
- Predictable concurrency behavior.

JavaScript, by design, carries limitations in efficiency and runtime guarantees. NodeJS relies on an event-driven model that works extremely well for many use cases, but under very high backend workloads or computationally intensive operations, it may not be the optimal choice. Some projects demand a language that combines:

- Native performance.
- Memory safety without garbage collection pauses.
- Strong static typing.
- Compile-time correctness guarantees.

Rust fulfills these requirements in a balanced and modern way.

A Practical Decision

After exploring Rust deeply across multiple directionssystems tools, infrastructure components, and performance-oriented applicationsI decided to write a specialized booklet focused on Web Backend Engineering.
The objective is not to criticize other ecosystems, but to provide an alternative path for projects that:

- Cannot tolerate performance unpredictability.
- Require higher safety guarantees.
- Demand architectural discipline from the ground up.

Rust offers that foundation.

Engineering, Not Trend Following

This booklet is written from an engineering standpoint, not from a trend-driven perspective.
I have worked through multiple technological waves over decades. Languages come and go.
Frameworks change. Ecosystems evolve.
What remains constant is the need for:

- Correctness.
- Performance.
- Security.
- Maintainability.

Rust aligns naturally with these enduring requirements.

Closing Thought

Coming from more than 30 years of C++ experience, I did not approach Rust as a replacement.
I approached it as a tool.
What I found was a language powerful enough for systems work, disciplined enough for safety-critical environments, and structured enough to support serious backend architectures.
This booklet is the result of that journey.
I hope it serves engineers who seek strong foundations for building modern, secure, and high-performance backend systems.

Ayman Alheraki

Preface

Why This Booklet Exists

Modern backend systems are no longer simple requestresponse applications. They are distributed, concurrent, security-sensitive, performance-critical systems that must operate reliably under unpredictable load.
Many backend tutorials focus on frameworks. This booklet focuses on engineering.
Rust Backend Engineering was written to demonstrate how Rust can be used not merely as a programming language, but as a foundation for building production-grade backend systems that are:

- Memory-safe without garbage collection,
- Concurrency-safe without data races,
- Explicit in error handling,
- Predictable in performance,
- Structured in architecture.

This booklet does not aim to teach Rust syntax from scratch. It assumes basic familiarity with Rust and focuses instead on applying Rust correctly in real backend environments.

The Engineering Perspective

Backend engineering is fundamentally about:

- Managing state safely,
- Controlling concurrency,
- Protecting sensitive data,
- Handling failures explicitly,
- Observing systems in production,
- Deploying reliably.

Rust aligns naturally with these goals.
Ownership and borrowing clarify lifecycle. The type system clarifies contracts. Traits define boundaries. The `Result` type forces explicit error propagation.
Instead of hiding complexity behind abstractions, Rust requires clarity. That clarity becomes a long-term advantage in backend systems that must survive real-world scale.

What This Booklet Covers

This booklet takes a structured journey through production backend development:

- Strongly typed routing and middleware pipelines,
- Authentication using Argon2 and JWT,
- Role-based authorization,
- Database integration using SQLx with compile-time query validation,

- Structured logging and request-scoped tracing,
- Health checks, metrics, and observability,
- Testing strategies from domain to integration,
- Deployment using optimized builds and containerization,
- Graceful shutdown and reverse proxy integration.

Each chapter is designed to build toward a cohesive production-ready architecture rather than isolated examples.

Design Philosophy

Several core principles guide this booklet:

- Separation of concerns between domain, API, and infrastructure.
- Explicit configuration through environment variables.
- Security-first design.
- Observability as a non-negotiable requirement.
- Fail-fast startup and graceful shutdown.
- Minimal hidden magic.

The goal is not to create the smallest example, but the most structurally sound example.

Audience

This booklet is intended for:

- Backend engineers transitioning to Rust.
- Systems programmers expanding into web services.
- Experienced developers seeking stronger guarantees in production systems.
- Architects evaluating Rust for backend infrastructure.

It is not framework marketing. It is not an introductory Rust tutorial. It is an engineering guide.

On Production Reality

Production systems expose weaknesses:

- Concurrency bugs surface under load.
- Poor error handling becomes downtime.
- Weak observability slows incident response.
- Insecure defaults lead to breaches.

Rust does not eliminate responsibility. But it significantly reduces accidental complexity and prevents entire categories of runtime failures.
When combined with disciplined architecture, it becomes a powerful tool for backend reliability.

A Practical Commitment

All examples in this booklet are written with production awareness:

- Release builds optimized for performance,
- Environment-based configuration,
- Structured logging,
- Secure password hashing,
- Token-based authentication,
- Explicit error mapping,
- Clean shutdown handling.

The objective is clarity, correctness, and sustainability.

Final Thought Before You Begin

Backend engineering is not about how quickly a service responds in development. It is about how reliably it behaves over years.
Rust enforces discipline at compile time. Architecture enforces discipline at design time.
Operational practices enforce discipline at runtime.
This booklet brings those three together.
Welcome to *Rust Backend Engineering*.

Introduction: Rusts Philosophy in Web Backend Development

1.1 Why Rust for Backend Systems

Backend systems are dominated by three constraints: **throughput**, **tail latency**, and **correctness under concurrency**. Rust was designed for exactly this class of software: long-running, concurrent, resource-sensitive services.

Rust fits backend realities

- **Memory safety without GC**: safe Rust prevents use-after-free, double-free, and data races. This directly targets common production outage classes.
- **Predictable performance**: no garbage collector pauses; you control allocation, copying, and lifetimes.
- **Concurrency by construction**: safe sharing requires synchronization types; the compiler blocks accidental shared-mutable races.
- **Explicit error handling**: errors are values. You design error boundaries and map failures to responses intentionally.

- **Modern async**: `async/await` enables highly concurrent I/O services while keeping code readable.

Windows-first dev workflow

On Windows, you want a stable toolchain and repeatable commands. Use PowerShell for day-to-day work:

```
# Verify toolchain
rustc --version
cargo --version

# Quality gates (run locally and in CI)
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all
```

1.2 Performance, Safety, and Concurrency

1.2.1 Performance: control data movement and allocations

In backends, performance issues are often caused by hidden copies, unnecessary allocations, or unbounded concurrency.

Example: avoid copies with shared buffers

```rust
use bytes::Bytes;

fn parse_payload(buf: Bytes) -> Result<(), &'static str> {
    if buf.is_empty() { return Err("empty"); }

    // Cheap clone: shares the same underlying buffer, no copy.
    let _view = buf.clone();

    // Parse using slices/views
    let _first = buf.get(0).copied().unwrap_or(0);
    Ok(())
}
```

1.2.2 Safety: enforce invariants with types

A production backend benefits when invariants are unrepresentable.

Example: IDs that cannot be zero

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct UserId(u64);

impl UserId {
    pub fn new(v: u64) -> Result<Self, &'static str> {
        if v == 0 { return Err("user id must be non-zero"); }
        Ok(Self(v))
    }
    pub fn get(self) -> u64 { self.0 }
}
```

1.2.3 Concurrency: bounded fan-out and structured cancellation

For backend calls (DB/cache/services), you need concurrency *with limits*. Spawn everything without bounds and you create self-inflicted overload.

Example: bounded parallelism with a semaphore

```rust
use std::sync::Arc;
use tokio::sync::Semaphore;
```

```
async fn fetch_one(id: u64) -> String {
    format!("item:{id}")
}

pub async fn fetch_many_bounded(ids: Vec<u64>, max_in_flight: usize) -> Vec<String> {
    let sem = Arc::new(Semaphore::new(max_in_flight));
    let mut tasks = Vec::with_capacity(ids.len());

    for id in ids {
        let permit = sem.clone().acquire_owned().await.unwrap();
        tasks.push(tokio::spawn(async move {
            let out = fetch_one(id).await;
            drop(permit);
            out
        }));
    }

    let mut out = Vec::new();
    for t in tasks {
        if let Ok(v) = t.await { out.push(v); }
    }
    out
}
```

Example: timeouts as policy

```
use std::time::Duration;

pub async fn call_with_timeout() -> Result<&'static str, &'static str> {
    let op = async { "ok" };
    tokio::time::timeout(Duration::from_millis(200), op)
        .await
        .map_err(|_| "timeout")
}
```

1.3 Express vs Axum: A Practical Conceptual Comparison

This is a conceptual comparison of two dominant styles:

- **Express**: dynamic middleware chain mutating a shared request/response.
- **Axum**: typed extraction + typed handlers, built on a layered service pipeline.

1.3.1 Handler contract: dynamic objects vs typed parameters

Express style (dynamic request augmentation)

```
// Conceptual Express example
import express from "express";
const app = express();

app.use(async (req: any, _res, next) => {
  // runtime attachment
  req.user = { id: 1, role: "admin" };
  next();
});

app.get("/me", (req: any, res) => {
  // handler assumes req.user exists
  res.json({ id: req.user.id, role: req.user.role });
});
```

Axum style (typed extraction into the function signature)

```
use axum::{
    extract::{Path, State},
    routing::get,
    Json, Router,
};
use serde::Serialize;
use std::sync::Arc;

#[derive(Clone)]
struct AppState {
    prefix: &'static str,
}

#[derive(Serialize)]
struct UserDto { id: u64, name: String }

async fn get_user(
    State(st): State<Arc<AppState>>,
    Path(id): Path<u64>,
) -> Json<UserDto> {
    Json(UserDto { id, name: format!("{}{}", st.prefix, id) })
}

pub fn router(state: Arc<AppState>) -> Router {
    Router::new()
        .route("/users/:id", get(get_user))
        .with_state(state)
}
```

Key difference: in Axum, what a handler needs is explicit and checked at compile time.

1.3.2 Error handling: implicit exceptions vs explicit results

Axum: return a typed error and map it into a response

```rust
use axum::{http::StatusCode, response::IntoResponse};

pub enum ApiError {
    BadRequest(&'static str),
    NotFound,
}

impl IntoResponse for ApiError {
    fn into_response(self) -> axum::response::Response {
        match self {
            ApiError::BadRequest(m) => (StatusCode::BAD_REQUEST, m).into_response(),
            ApiError::NotFound => (StatusCode::NOT_FOUND, "not found").into_response(),
        }
    }
}

async fn handler() -> Result<&'static str, ApiError> {
    Err(ApiError::BadRequest("missing field"))
}
```

1.4 From Dynamic Middleware to Typed Pipelines

1.4.1 The migration mindset

Dynamic middleware often relies on convention:

- attach data to `req` at runtime,
- enforce correctness by middleware ordering,

- discover missing fields at runtime.

Typed pipelines aim for:

- **typed context** stored and extracted safely,
- **composable layers** with explicit behavior,
- **thin HTTP adapters** over stable domain logic.

1.4.2 Typed request context using extensions

Middleware inserts a typed value

```rust
use axum::{http::Request, middleware::Next, response::Response};

#[derive(Clone, Debug)]
pub struct User {
    pub id: u64,
    pub role: &'static str,
}

pub async fn auth<B>(mut req: Request<B>, next: Next<B>) -> Response {
    // Replace with real auth verification
    let user = User { id: 1, role: "admin" };
    req.extensions_mut().insert(user);
    next.run(req).await
}
```

Handler extracts the typed value

```rust
use axum::{extract::Extension, Json};
use serde::Serialize;
```

```
#[derive(Serialize)]
struct Me { id: u64, role: &'static str }

pub async fn me(Extension(user): Extension<crate::auth::User>) -> Json<Me> {
    Json(Me { id: user.id, role: user.role })
}
```

This replaces "maybe `req.user` exists" with "the request either contains a `User` or it does not," and the extractor enforces it.

1.4.3 Composing middleware as layers (pipeline over services)

A typical Axum stack uses layers to add cross-cutting concerns (timeouts, tracing, request IDs). This model is deterministic and testable.

Example: build a production-like HTTP stack

```
use std::time::Duration;
use axum::{routing::get, Router};
use tower::ServiceBuilder;
use tower_http::{
    request_id::{MakeRequestUuid, SetRequestIdLayer},
    timeout::TimeoutLayer,
    trace::TraceLayer,
};

async fn health() -> &'static str { "ok" }

pub fn app() -> Router {
    let stack = ServiceBuilder::new()
        .layer(SetRequestIdLayer::new(
            axum::http::HeaderName::from_static("x-request-id"),
            MakeRequestUuid,
        ))
        .layer(TraceLayer::new_for_http())
        .layer(TimeoutLayer::new(Duration::from_secs(3)));

    Router::new()
        .route("/health", get(health))
        .layer(stack)
}
```

1.4.4 Windows run commands (PowerShell)

```
# Run a binary crate (example name: app)
cargo run -p app

# Build release (Windows .exe output)
cargo build -p app --release

# Optional: set env vars for current session
$env:RUST_LOG = "info"
cargo run -p app
```

What you should internalize from this chapter

- Rust backend engineering is about **reliability under concurrency** as much as it is about speed.
- Axum encourages **typed handlers and typed extraction**, reducing runtime coupling found in dynamic middleware chains.
- A layered pipeline makes cross-cutting concerns (timeouts, tracing, IDs) **explicit, composable, and testable**.

Environment Setup and Project Architecture

2.1 Installing Rust and Tooling

2.1.1 Install Rust on Windows using `rustup`

On Windows, the most reliable and team-friendly setup is `rustup` with the MSVC toolchain (native Windows linker/toolchain). You do not need the full Visual Studio IDE, but you do need the C++ Build Tools for linking.

Install `rustup` via `winget` (PowerShell)

```
# Install rustup (Rust toolchain manager)
winget install -e --id Rustlang.Rustup

# Close/reopen PowerShell after installation (PATH refresh), then:
rustup --version
```

Initialize the MSVC toolchain

```
# Install stable and make it the default (MSVC)
rustup toolchain install stable
rustup default stable-msvc

# Verify
rustc --version
cargo --version
```

Install essential components

```
rustup component add rustfmt clippy rust-src

# Update toolchains/components later
rustup update
```

2.1.2 Editor tooling (Windows-first)

Recommended baseline:

- VS Code + Rust Analyzer extension
- `rustfmt` for formatting, `clippy` for linting

Useful per-project commands (use in CI too):

```
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all
```

2.1.3 Pin the toolchain per repository

Create `rust-toolchain.toml` in the repository root so all developers and CI use the same toolchain:

```
[toolchain]
channel = "stable"
components = ["rustfmt", "clippy", "rust-src"]
```

2.2 Creating a Production-Ready Cargo Project

2.2.1 Why a workspace

A production backend should be a workspace with multiple crates to enforce boundaries:

- easier scaling and testing,
- clearer dependency direction,
- safer refactoring.

2.2.2 Create a Windows-native workspace (PowerShell)

```
mkdir RustBackendEngineering
cd RustBackendEngineering

# Workspace root Cargo.toml
@"
[workspace]
resolver = "2"
members = ["crates/app", "crates/api", "crates/domain", "crates/infrastructure"]

[workspace.package]
edition = "2024"
license = "MIT OR Apache-2.0"
"@ | Set-Content -Encoding UTF8 Cargo.toml

# Create crates
cargo new crates/app --bin
cargo new crates/api --lib
cargo new crates/domain --lib
cargo new crates/infrastructure --lib
```

2.2.3 Production-minded build profiles

Add these to the workspace root `Cargo.toml`:

```toml
[profile.dev]
debug = true

[profile.release]
lto = true
codegen-units = 1
panic = "abort"
strip = "symbols"
```

2.2.4 Centralize dependency versions at workspace level

Add `[workspace.dependencies]` to the workspace root `Cargo.toml`:

```toml
[workspace.dependencies]
tokio = { version = "1", features = ["rt-multi-thread", "macros", "signal"] }
axum = "0.7"
tower = "0.5"
tower-http = { version = "0.6", features = ["trace", "request-id", "timeout"] }

serde = { version = "1", features = ["derive"] }
serde_json = "1"

tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter"] }

config = "0.14"
dotenvy = "0.15"
```

Then use workspace deps inside each crate.

2.3 Clean Project Structure

2.3.1 Recommended Windows-friendly layout

```
RustBackendEngineering\
  Cargo.toml
  rust-toolchain.toml
  .env
  config\
    local.toml
    production.toml
  crates\
    app\
      Cargo.toml
      src\main.rs
    api\
      Cargo.toml
      src\lib.rs
      src\routes.rs
      src\handlers\mod.rs
      src\handlers\health.rs
      src\errors.rs
    domain\
      Cargo.toml
      src\lib.rs
      src\models\mod.rs
      src\models\user.rs
      src\services\mod.rs
      src\services\user_service.rs
      src\errors.rs
    infrastructure\
      Cargo.toml
      src\lib.rs
      src\config.rs
      src\logging.rs
      src\shutdown.rs
```

2.3.2 Dependency direction (rule)

Keep a strict direction to avoid framework leakage:

- `domain` depends on nothing else in the workspace
- `infrastructure` depends on third-party libs for config/logging/adapters
- `api` depends on `domain` and may use `infrastructure` utilities
- app depends on all and wires them together

2.4 Modular Design: app / api / domain / infrastructure

2.4.1 Domain: invariants and pure logic (no HTTP, no config)

```rust
//// crates/domain/src/models/user.rs
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct UserId(u64);

impl UserId {
    pub fn new(v: u64) -> Result<Self, &'static str> {
        if v == 0 { return Err("user id must be non-zero"); }
        Ok(Self(v))
    }
    pub fn get(&self) -> u64 { self.0 }
}

#[derive(Debug, Clone)]
```

```rust
pub struct User {
    pub id: UserId,
    pub name: String,
}
```

```rust
//// crates/domain/src/services/user_service.rs
use crate::models::user::{User, UserId};

pub fn build_user(id: u64, name: &str) -> Result<User, &'static str> {
    let id = UserId::new(id)?;
    let name = name.trim();
    if name.is_empty() { return Err("name is empty"); }
    if name.len() > 64 { return Err("name too long"); }
    Ok(User { id, name: name.to_string() })
}
```

2.4.2 API: Axum routes and handlers (thin adapters)

```toml
# crates/api/Cargo.toml
[package]
name = "api"
version = "0.1.0"
edition.workspace = true

[dependencies]
axum.workspace = true
serde.workspace = true
domain = { path = "../domain" }
```

```rust
//// crates/api/src/routes.rs
use axum::{routing::get, Router};

pub fn router() -> Router {
    Router::new()
```

```
        .route("/health", get(crate::handlers::health::health))
        .route("/users/:id", get(crate::handlers::users::get_user))
}
```

```
//// crates/api/src/handlers/health.rs
use axum::response::IntoResponse;

pub async fn health() -> impl IntoResponse {
    "ok"
}
```

```
//// crates/api/src/handlers/users.rs
use axum::{extract::Path, Json};
use serde::Serialize;

#[derive(Serialize)]
pub struct UserDto {
    pub id: u64,
    pub name: String,
}

pub async fn get_user(Path(id): Path<u64>) -> Result<Json<UserDto>,
↪   (axum::http::StatusCode, &'static str)> {
    let user = domain::services::user_service::build_user(id, "WindowsUser")
        .map_err(|_| (axum::http::StatusCode::BAD_REQUEST, "invalid user"))?;

    Ok(Json(UserDto {
        id: user.id.get(),
        name: user.name,
    }))
}
```

2.4.3 Infrastructure: logging, config, and system helpers

```toml
# crates/infrastructure/Cargo.toml
[package]
name = "infrastructure"
version = "0.1.0"
edition.workspace = true

[dependencies]
tracing.workspace = true
tracing-subscriber.workspace = true
config.workspace = true
dotenvy.workspace = true
serde.workspace = true
```

```rust
//// crates/infrastructure/src/logging.rs
use tracing_subscriber::{fmt, EnvFilter};

pub fn init_logging(default_level: &str) {
    let filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new(default_level));

    fmt()
        .with_env_filter(filter)
        .with_target(false)
        .with_thread_names(true)
        .with_thread_ids(true)
        .compact()
        .init();
}
```

2.5 Configuration Management

2.5.1 Windows-first environment setup

On Windows, environment variables are commonly set either:

- for the current PowerShell session: `$env:KEY="value"`
- persistently for the user: `setx KEY value` (takes effect in new terminals)

Create config files

```
mkdir config

@"
[server]
host = "0.0.0.0"
port = 8080

[log]
level = "info"

[features]
enable_dev_routes = true
"@ | Set-Content -Encoding UTF8 config\local.toml

@"
[server]
host = "0.0.0.0"
port = 8080

[log]
level = "info"
```

```
[features]
enable_dev_routes = false
"@ | Set-Content -Encoding UTF8 config\production.toml
```

Local .env for Windows development

```
@"
APP_ENV=local
APP_SERVER__PORT=8081
APP_LOG__LEVEL=debug
"@ | Set-Content -Encoding UTF8 .env
```

Config loader (typed) in infrastructure

```rust
//// crates/infrastructure/src/config.rs
use serde::Deserialize;

#[derive(Debug, Clone, Deserialize)]
pub struct AppConfig {
    pub server: ServerConfig,
    pub log: LogConfig,
    pub features: FeaturesConfig,
}

#[derive(Debug, Clone, Deserialize)]
pub struct ServerConfig {
    pub host: String,
    pub port: u16,
}

#[derive(Debug, Clone, Deserialize)]
pub struct LogConfig {
    pub level: String,
}

#[derive(Debug, Clone, Deserialize)]
pub struct FeaturesConfig {
    pub enable_dev_routes: bool,
}

pub fn load() -> Result<AppConfig, config::ConfigError> {
    // Windows: dotenv is optional; ignore missing .env
    let _ = dotenvy::dotenv();

    let env = std::env::var("APP_ENV").unwrap_or_else(|_| "local".to_string());
    let file = format!("config/{env}.toml");

    let cfg = config::Config::builder()
        .add_source(config::File::with_name("config/default").required(false))
        .add_source(config::File::with_name(&file).required(false))
        .add_source(config::Environment::with_prefix("APP").separator("__"))
        .build()?;

    cfg.try_deserialize()
}
```

2.5.2 App crate: compose everything and run on Windows

```toml
# crates/app/Cargo.toml
[package]
name = "app"
version = "0.1.0"
edition.workspace = true

[dependencies]
tokio.workspace = true
```

```
axum.workspace = true
tower.workspace = true
tower-http.workspace = true

tracing.workspace = true
tracing-subscriber.workspace = true

api = { path = "../api" }
domain = { path = "../domain" }
infrastructure = { path = "../infrastructure" }
```

```
//// crates/app/src/main.rs
use std::{net::SocketAddr, time::Duration};
use axum::Router;
use tower::ServiceBuilder;
use tower_http::{timeout::TimeoutLayer, trace::TraceLayer};

fn build_app(enable_dev_routes: bool) -> Router {
    let mut r = api::routes::router();

    if enable_dev_routes {
        r = r.route("/dev/ping", axum::routing::get(|| async { "pong" }));
    }

    let stack = ServiceBuilder::new()
        .layer(TraceLayer::new_for_http())
        .layer(TimeoutLayer::new(Duration::from_secs(5)));

    r.layer(stack)
}

#[tokio::main]
async fn main() {
    let cfg = infrastructure::config::load().expect("config load failed");
    infrastructure::logging::init_logging(&cfg.log.level);

    let app = build_app(cfg.features.enable_dev_routes);

    let addr: SocketAddr = format!("{}:{}", cfg.server.host, cfg.server.port)
        .parse()
        .expect("invalid bind address");

    let listener = tokio::net::TcpListener::bind(addr).await.expect("bind failed");
    tracing::info!("listening on {}", listener.local_addr().unwrap());

    axum::serve(listener, app).await.expect("server error");
}
```

2.5.3 Run on Windows (PowerShell)

```
# from workspace root
cargo run -p app

# set environment for current session (optional)
$env:APP_ENV = "local"
$env:APP_LOG__LEVEL = "debug"
cargo run -p app

# persistent (new terminals only)
setx APP_ENV local
setx APP_LOG__LEVEL debug
```

Chapter outcomes

At the end of this chapter, you have a Windows-native Rust backend workspace that:

- installs and pins Rust toolchains consistently,

- enforces formatting/linting/testing habits,
- uses a clean crate boundary: `app/api/domain/infrastructure`,
- loads configuration in a typed, layered, environment-driven way suitable for Windows deployments.

Routing in Rust: Express-Style but Strongly Typed

3.1 Building the Router Tree

Axum routing feels familiar to Express users (paths, methods, middleware), but the structure is strongly typed and composition-driven: you build a `Router` tree by nesting sub-routers and mounting them under prefixes.

Workspace-style structure

```
crates/
  app/                 # binary: composition root
  api/                 # axum router tree + handlers + AppError
  domain/              # business logic (no HTTP)
  infrastructure/      # config/logging/shutdown
```

Router tree composition (api crate)

```rust
//// crates/api/src/routes.rs
use axum::{routing::{get, post}, Router};
use std::sync::Arc;
use crate::{handlers, state::AppState};

pub fn router(state: Arc<AppState>) -> Router {
    let v1 = Router::new()
        .nest("/users", users_routes())
        .nest("/admin", admin_routes());

    Router::new()
        .route("/health", get(handlers::health::health))
        .nest("/v1", v1)
        .with_state(state)
}

fn users_routes() -> Router {
    Router::new()
        .route("/", post(handlers::users::create_user))
        .route("/:id", get(handlers::users::get_user))
        .route("/:id", post(handlers::users::update_user))
}

fn admin_routes() -> Router {
    Router::new()
        .route("/metrics", get(handlers::admin::metrics))
}
```

Application state (shared dependencies)

Keep DB pools, clients, and config in one shared state object.

```
//// crates/api/src/state.rs
#[derive(Clone)]
pub struct AppState {
```

```rust
    pub service_name: &'static str,
}

impl AppState {
    pub fn new(service_name: &'static str) -> Self {
        Self { service_name }
    }
}
```

App crate wiring (Windows-friendly)

```rust
//// crates/app/src/main.rs
use std::{net::SocketAddr, sync::Arc};

#[tokio::main]
async fn main() {
    infrastructure::logging::init_logging("info");

    let state = Arc::new(api::state::AppState::new("rust-backend-engineering"));
    let app = api::routes::router(state);

    let addr: SocketAddr = "0.0.0.0:8080".parse().unwrap();
    let listener = tokio::net::TcpListener::bind(addr).await.unwrap();

    axum::serve(listener, app).await.unwrap();
}
```

3.2 Route Parameters and Query Extraction

Axum uses **extractors** to decode request data into typed handler parameters.

3.2.1 Path parameters

```rust
//// crates/api/src/handlers/users.rs (excerpt)
use axum::extract::Path;

pub async fn get_user(Path(id): Path<u64>) -> &'static str {
    let _ = id;
    "ok"
}
```

3.2.2 Multiple parameters

```rust
use axum::extract::Path;

pub async fn get_user_post(Path((user_id, post_id)): Path<(u64, u64)>) -> String {
    format!("user={user_id} post={post_id}")
}
```

3.2.3 Query parameters (typed)

```rust
use axum::extract::Query;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
pub struct ListUsersQuery {
    pub page: Option<u32>,
    pub page_size: Option<u32>,
    pub q: Option<String>,
}

pub async fn list_users(Query(q): Query<ListUsersQuery>) -> String {
    let page = q.page.unwrap_or(1);
    let page_size = q.page_size.unwrap_or(20);
    let term = q.q.unwrap_or_default();
    format!("page={page} size={page_size} q={term}")
}
```

3.3 JSON Request and Response Handling

3.3.1 JSON request bodies

Axum provides `Json<T>` where `T: Deserialize`. You get typed payloads and automatic validation of JSON syntax.

```
use axum::Json;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
pub struct CreateUserRequest {
    pub name: String,
    pub email: String,
}

pub async fn create_user(Json(req): Json<CreateUserRequest>) -> String {
    format!("created: {} <{}>", req.name, req.email)
}
```

3.3.2 JSON responses

Return `Json<T>` where `T: Serialize`.

```
use axum::Json;
use serde::Serialize;

#[derive(Debug, Serialize)]
```

```rust
pub struct UserResponse {
    pub id: u64,
    pub name: String,
    pub email: String,
}

pub async fn get_user_json() -> Json<UserResponse> {
    Json(UserResponse {
        id: 1,
        name: "WindowsUser".to_string(),
        email: "user@example.com".to_string(),
    })
}
```

3.3.3 Request validation pattern (minimal but strict)

Rust encourages enforcing invariants early and returning typed errors.

```rust
fn validate_create(req: &CreateUserRequest) -> Result<(), &'static str> {
    let name = req.name.trim();
    if name.is_empty() { return Err("name is empty"); }
    if name.len() > 64 { return Err("name too long"); }
    if !req.email.contains('@') { return Err("email invalid"); }
    Ok(())
}
```

3.4 Status Codes and Custom Responses

3.4.1 Returning a status code with JSON

```rust
use axum::{http::StatusCode, Json};
use serde::Serialize;
```

```
#[derive(Serialize)]
pub struct Created {
    pub id: u64,
}

pub async fn create_ok() -> (StatusCode, Json<Created>) {
    (StatusCode::CREATED, Json(Created { id: 42 }))
}
```

3.4.2 Custom headers and bodies

```
use axum::{
    http::{HeaderMap, HeaderValue, StatusCode},
    response::IntoResponse,
};

pub async fn custom_response() -> impl IntoResponse {
    let mut headers = HeaderMap::new();
    headers.insert("x-service", HeaderValue::from_static("rust-backend-engineering"));
    (StatusCode::OK, headers, "ok")
}
```

3.4.3 Returning `Response` for full control

```
use axum::{
    body::Body,
    http::{Response, StatusCode},
};

pub async fn raw_response() -> Response<Body> {
    Response::builder()
        .status(StatusCode::ACCEPTED)
        .header("content-type", "text/plain; charset=utf-8")
        .body(Body::from("accepted"))
        .unwrap()
}
```

3.5 Centralized Error Handling (AppError Pattern)

A production backend should centralize error mapping so handlers can return `Result<T, AppError>` and the mapping to HTTP responses remains consistent.

3.5.1 Define `AppError` and a shared `Result` type

```
//// crates/api/src/error.rs
use axum::{
    extract::rejection::JsonRejection,
    http::StatusCode,
    response::{IntoResponse, Response},
    Json,
};
use serde::Serialize;

pub type AppResult<T> = Result<T, AppError>;

#[derive(Debug)]
pub enum AppError {
    BadRequest(&'static str),
    NotFound,
    Conflict(&'static str),
    Json(JsonRejection),
    Internal(&'static str),
}
```

```
#[derive(Serialize)]
struct ErrorBody {
    code: u16,
    message: String,
}

impl AppError {
    fn to_body(&self, status: StatusCode, message: String) -> (StatusCode, Json<ErrorBody>)
    ↪  {
        (
            status,
            Json(ErrorBody {
                code: status.as_u16(),
                message,
            }),
        )
    }
}

impl From<JsonRejection> for AppError {
    fn from(e: JsonRejection) -> Self {
        AppError::Json(e)
    }
}

impl IntoResponse for AppError {
    fn into_response(self) -> Response {
        match self {
            AppError::BadRequest(msg) =>
                self.to_body(StatusCode::BAD_REQUEST, msg.to_string()).into_response(),

            AppError::NotFound =>
              self.to_body(StatusCode::NOT_FOUND, "not found".to_string()).into_response(),
            AppError::Conflict(msg) =>
                self.to_body(StatusCode::CONFLICT, msg.to_string()).into_response(),

            AppError::Json(rej) => {
                // Keep message safe for clients; avoid leaking internals.
                let msg = format!("invalid JSON: {}", rej.body_text());
                self.to_body(StatusCode::BAD_REQUEST, msg).into_response()
            }

            AppError::Internal(msg) =>
                self.to_body(StatusCode::INTERNAL_SERVER_ERROR,
                ↪  msg.to_string()).into_response(),
        }
    }
}
```

3.5.2 Use `AppResult` in handlers

```
//// crates/api/src/handlers/users.rs
use axum::{extract::Path, Json};
use serde::{Deserialize, Serialize};

use crate::error::{AppError, AppResult};

#[derive(Debug, Deserialize)]
pub struct CreateUserRequest {
    pub name: String,
    pub email: String,
}

#[derive(Debug, Serialize)]
pub struct UserResponse {
    pub id: u64,
    pub name: String,
    pub email: String,
}

fn validate(req: &CreateUserRequest) -> AppResult<()> {
    let name = req.name.trim();
    if name.is_empty() { return Err(AppError::BadRequest("name is empty")); }
    if name.len() > 64 { return Err(AppError::BadRequest("name too long")); }
    if !req.email.contains('@') { return Err(AppError::BadRequest("email invalid")); }
    Ok(())
}

pub async fn create_user(Json(req): Json<CreateUserRequest>) ->
↪   AppResult<Json<UserResponse>> {
    validate(&req)?;

    // Replace with DB insert
    let user = UserResponse { id: 100, name: req.name, email: req.email };
    Ok(Json(user))
}

pub async fn get_user(Path(id): Path<u64>) -> AppResult<Json<UserResponse>> {
    if id == 0 {
        return Err(AppError::BadRequest("id must be non-zero"));
    }
    if id == 404 {
        return Err(AppError::NotFound);
    }

    Ok(Json(UserResponse {
        id,
        name: format!("user-{id}"),
        email: format!("user{id}@example.com"),
    }))
}

pub async fn update_user(Path(id): Path<u64>, Json(req): Json<CreateUserRequest>)
    -> AppResult<Json<UserResponse>>
{
    validate(&req)?;

    if id == 1 && req.email == "taken@example.com" {
        return Err(AppError::Conflict("email already exists"));
    }

    Ok(Json(UserResponse { id, name: req.name, email: req.email }))
}
```

3.5.3 Route table using these handlers

```
//// crates/api/src/routes.rs (minimal excerpt)
use axum::{routing::{get, post}, Router};

pub fn users_routes() -> Router {
    Router::new()
        .route("/", post(crate::handlers::users::create_user))
        .route("/:id", get(crate::handlers::users::get_user))
        .route("/:id", post(crate::handlers::users::update_user))
}
```

3.6 Windows build and run commands (PowerShell)

```
# From workspace root
cargo run -p app
```

```
# Release build produces a Windows .exe
cargo build -p app --release

# Quality gates
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all
```

Chapter outcomes

- You can build a router tree with nested routers, Express-style.
- You can extract path/query/body data as typed parameters.
- You can return JSON, status codes, headers, or raw responses explicitly.
- You can centralize failures using a consistent `AppError` pattern for the whole API.

Middleware Pipeline and Request Lifecycle

4.1 Logging Middleware (Tracing)

Production backends need **structured**, **low-overhead** logs correlated to requests. In Rust, the standard pattern is:

- initialize `tracing` subscriber once in `main`,
- use HTTP tracing middleware to emit request spans and response events,
- include request identifiers in the span so every log line is searchable.

Logging initialization (Windows, `infrastructure` crate)

```
//// crates/infrastructure/src/logging.rs
use tracing_subscriber::{fmt, EnvFilter};

pub fn init_logging(default_level: &str) {
    let filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new(default_level));

    fmt()
        .with_env_filter(filter)
        .with_target(false)
        .with_thread_names(true)
        .with_thread_ids(true)
        .compact()
        .init();
}
```

HTTP tracing layer (API pipeline)

```
use tower_http::trace::TraceLayer;

fn tracing_layer() ->
↪   TraceLayer<tower_http::classify::SharedClassifier<tower_http::classify::ServerErrorsAsFailures
↪   {
    TraceLayer::new_for_http()
}
```

4.2 Request ID and Correlation Handling

A request ID is the simplest correlation primitive:

- accept an incoming `x-request-id` if present (from gateway),
- otherwise generate one,
- attach it to the response,
- ensure it appears in logs/traces.

Request ID middleware stack

```
use axum::http::HeaderName;
use tower::ServiceBuilder;
use tower_http::request_id::{MakeRequestUuid, SetRequestIdLayer, PropagateRequestIdLayer};
```

```
fn request_id_layers() -> ServiceBuilder<tower::layer::util::Identity> {
    let name = HeaderName::from_static("x-request-id");

    ServiceBuilder::new()
        // Generate one if missing
        .layer(SetRequestIdLayer::new(name.clone(), MakeRequestUuid))
        // Copy request id into response
        .layer(PropagateRequestIdLayer::new(name))
}
```

Using request ID in application logs

In handlers, log normally; the request span will carry the correlation headers:

```
use tracing::info;

pub async fn handler() -> &'static str {
    info!("handler entered");
    "ok"
}
```

4.3 CORS Configuration

CORS is a browser security mechanism. Your backend should:

- restrict origins for production,
- explicitly allow methods/headers you actually use,
- optionally allow credentials only when required.

Strict CORS example (recommended for production)

```
use tower_http::cors::{CorsLayer, Any};
use axum::http::{HeaderValue, Method};

fn cors_layer_strict() -> CorsLayer {
    CorsLayer::new()
        .allow_origin([
            HeaderValue::from_static("https://example.com"),
            HeaderValue::from_static("https://admin.example.com"),
        ])
        .allow_methods([Method::GET, Method::POST, Method::PUT, Method::DELETE])
        .allow_headers(Any)
}
```

Permissive CORS (local development only)

```
use tower_http::cors::CorsLayer;

fn cors_layer_dev() -> CorsLayer {
    CorsLayer::permissive()
}
```

4.4 Rate Limiting and Timeouts

Two common production protections:

- **timeouts**: cap maximum time per request (or per downstream call),
- **rate limiting**: cap request rate to prevent overload and abuse.

4.4.1 Timeout layer

```rust
use std::time::Duration;
use tower_http::timeout::TimeoutLayer;

fn timeout_layer() -> TimeoutLayer {
    TimeoutLayer::new(Duration::from_secs(3))
}
```

4.4.2 Rate limiting layer

`RateLimitLayer` is a simple global limiter for a service instance:

- good for baseline protection,
- not user/IP-specific unless you build keyed limiting separately.

```rust
use std::time::Duration;
use tower::limit::RateLimitLayer;

fn rate_limit_layer() -> RateLimitLayer {
    // Allow N requests per time window for this service instance.
    RateLimitLayer::new(100u64, Duration::from_secs(1))
}
```

4.4.3 Concurrency limiting (often more important than rate)

Concurrency limiting prevents resource exhaustion under slow downstream dependencies:

```rust
use tower::limit::ConcurrencyLimitLayer;

fn concurrency_limit_layer() -> ConcurrencyLimitLayer {
    ConcurrencyLimitLayer::new(256)
}
```

4.4.4 Handler-level timeouts for downstream calls

Even with a global request timeout, also enforce downstream call timeouts:

```rust
use std::time::Duration;

pub async fn call_dependency() -> Result<&'static str, &'static str> {
    let op = async {
        // replace with DB/HTTP call
        "ok"
    };

    tokio::time::timeout(Duration::from_millis(200), op)
        .await
        .map_err(|_| "dependency timeout")
}
```

4.5 Compression and Security Headers

4.5.1 Compression

Compression should usually be enabled for text payloads (JSON, HTML) and disabled for already-compressed formats.

```rust
use tower_http::compression::CompressionLayer;

fn compression_layer() -> CompressionLayer {
    CompressionLayer::new()
}
```

4.5.2 Security headers

A practical baseline for many APIs:

- `x-content-type-options: nosniff`
- `x-frame-options: DENY` (or `SAMEORIGIN` for embedded admin UIs)
- `referrer-policy: no-referrer` (or stricter as needed)

```rust
use axum::http::{HeaderName, HeaderValue};
use tower_http::set_header::SetResponseHeaderLayer;

fn security_headers_layers() -> tower::ServiceBuilder<tower::layer::util::Identity> {
    tower::ServiceBuilder::new()
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("x-content-type-options"),
            HeaderValue::from_static("nosniff"),
        ))
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("x-frame-options"),
            HeaderValue::from_static("deny"),
        ))
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("referrer-policy"),
            HeaderValue::from_static("no-referrer"),
        ))
}
```

4.6 Putting It All Together: The Middleware Pipeline

The pipeline should be built once (composition root) and applied to the router. Keep ordering intentional:

- request ID early (so all logs include it),
- tracing early (wraps everything),

- security headers and CORS,
- limits and timeouts,
- compression near the end (response shaping).

Full pipeline composition (Windows-ready)

```rust
//// crates/app/src/main.rs
use std::{net::SocketAddr, sync::Arc, time::Duration};

use axum::Router;
use tower::ServiceBuilder;

use axum::http::{HeaderName, Method};
use tower_http::{
    classify::ServerErrorsAsFailures,
    cors::{Any, CorsLayer},
    request_id::{MakeRequestUuid, SetRequestIdLayer, PropagateRequestIdLayer},
    timeout::TimeoutLayer,
    trace::TraceLayer,
    compression::CompressionLayer,
    set_header::SetResponseHeaderLayer,
};
use tower::limit::{ConcurrencyLimitLayer, RateLimitLayer};

#[tokio::main]
async fn main() {
    infrastructure::logging::init_logging("info");

    let state = Arc::new(api::state::AppState::new("rust-backend-engineering"));
    let app = api::routes::router(state);
    let request_id = HeaderName::from_static("x-request-id");

    // Choose strict CORS in production.
    let cors = CorsLayer::new()
        .allow_origin(Any)
        .allow_methods([Method::GET, Method::POST, Method::PUT, Method::DELETE])
        .allow_headers(Any);

    let middleware = ServiceBuilder::new()
        .layer(SetRequestIdLayer::new(request_id.clone(), MakeRequestUuid))
        .layer(PropagateRequestIdLayer::new(request_id))
        .layer(TraceLayer::new_for_http().on_failure(ServerErrorsAsFailures::new()))
        .layer(SetResponseHeaderLayer::overriding(
            axum::http::HeaderName::from_static("x-content-type-options"),
            axum::http::HeaderValue::from_static("nosniff"),
        ))
        .layer(SetResponseHeaderLayer::overriding(
            axum::http::HeaderName::from_static("x-frame-options"),
            axum::http::HeaderValue::from_static("deny"),
        ))
        .layer(cors)
        .layer(ConcurrencyLimitLayer::new(256))
        .layer(RateLimitLayer::new(100u64, Duration::from_secs(1)))
        .layer(TimeoutLayer::new(Duration::from_secs(3)))
        .layer(CompressionLayer::new());

    let app = app.layer(middleware);

    let addr: SocketAddr = "0.0.0.0:8080".parse().unwrap();
    let listener = tokio::net::TcpListener::bind(addr).await.unwrap();

    axum::serve(listener, app).await.unwrap();
}
```

4.7 Windows Commands (PowerShell)

```
# Run (development)
cargo run -p app

# Release build produces .exe
cargo build -p app --release

# Set log level for current PowerShell session
$env:RUST_LOG = "info"
cargo run -p app

# Quality gates
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all
```

Chapter outcomes

- You built a deterministic middleware pipeline for request lifecycle management.
- You added correlation (request ID), structured logging (tracing), and policy layers (CORS, limits, timeouts).
- You enabled compression and baseline security headers suitable for modern API services.

Production-Grade Security Fundamentals

5.1 HTTPS and Reverse Proxy Considerations

Most production deployments terminate TLS at a reverse proxy (or ingress) and forward traffic to your Rust service over a private network. Your application must be proxy-aware to preserve security guarantees and correct behavior.

Recommended production topology

- **Client** → **Reverse Proxy / Load Balancer** (TLS termination, WAF, rate limiting)
- → **Rust Service** (HTTP on internal network)

What the proxy must provide

- TLS termination with modern cipher suites and certificate rotation
- forwarding headers:
  - `X-Forwarded-Proto` (http/https)
  - `X-Forwarded-For` (client IP chain)
  - `X-Forwarded-Host` (original Host)

- request buffering and size limits
- optional WAF rules, bot protection, and global rate limits

Application-side rules

- Never trust forwarded headers from the open internet. Trust them only if traffic comes from your proxy network.
- Treat `X-Forwarded-Proto=https` as authoritative only from trusted proxy IP ranges.
- Use `Host` and scheme carefully when generating absolute URLs (redirects, links).

Example: proxy-aware scheme detection (trusted proxy only)

```rust
use axum::{extract::ConnectInfo, http::Request, middleware::Next, response::Response};
use std::net::SocketAddr;

#[derive(Clone, Copy, Debug)]
pub enum Scheme { Http, Https }

#[derive(Clone, Debug)]
pub struct RequestMeta {
    pub scheme: Scheme,
    pub client_addr: SocketAddr,
}

// Example trust rule: only accept forwarded headers when remote peer is in a known range.
// Replace with your real CIDR checks or explicit proxy IP allowlist.
fn is_trusted_proxy(peer: &SocketAddr) -> bool {
    peer.ip().is_loopback()
}
```

```
pub async fn detect_scheme<B>(
    ConnectInfo(peer): ConnectInfo<SocketAddr>,
    mut req: Request<B>,
    next: Next<B>,
) -> Response {
    let scheme = if is_trusted_proxy(&peer) {
        match req.headers().get("x-forwarded-proto").and_then(|v| v.to_str().ok()) {
            Some("https") => Scheme::Https,
            _ => Scheme::Http,
        }
    } else {
        Scheme::Http
    };

    req.extensions_mut().insert(RequestMeta { scheme, client_addr: peer });
    next.run(req).await
}
```

Windows run note

On Windows, you will typically run the Rust service behind IIS ARR, Nginx, HAProxy, or a cloud load balancer. The same trust rules apply.

5.2 Secure HTTP Headers (HSTS, CSP, etc.)

Security headers should be applied consistently (preferably at the proxy) and optionally duplicated at the application layer for defense-in-depth.

Header policy overview

- **HSTS** (`Strict-Transport-Security`): force HTTPS usage by browsers.

- **CSP** (`Content-Security-Policy`): prevent script injection by restricting sources.
- **X-Content-Type-Options: nosniff**: blocks MIME sniffing.
- **X-Frame-Options** or CSP `frame-ancestors`: prevents clickjacking.
- **Referrer-Policy**: controls referrer leakage.
- **Permissions-Policy**: limits browser features (camera, geolocation, etc.) if relevant.

CSP guidance for APIs

If you only serve JSON (no HTML), CSP is typically less critical. If you serve admin pages or documentation pages from the same origin, CSP becomes important.

Axum security headers (application layer)

```rust
use axum::http::{HeaderName, HeaderValue};
use tower::ServiceBuilder;
use tower_http::set_header::SetResponseHeaderLayer;

pub fn security_headers() -> ServiceBuilder<tower::layer::util::Identity> {
    ServiceBuilder::new()
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("x-content-type-options"),
            HeaderValue::from_static("nosniff"),
        ))
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("referrer-policy"),
            HeaderValue::from_static("no-referrer"),
        ))
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("permissions-policy"),
            HeaderValue::from_static("geolocation=(), camera=(), microphone=()"),
        ))
        // Prefer CSP frame-ancestors over X-Frame-Options when you need fine control.
        .layer(SetResponseHeaderLayer::overriding(
            HeaderName::from_static("content-security-policy"),
            HeaderValue::from_static("default-src 'none'; frame-ancestors 'none'"),
        ))
}
```

HSTS (only when you are truly HTTPS-only)

HSTS must be enabled only when your domain is reliably HTTPS-only. Enable it at the proxy, and optionally at the app:

```
use axum::http::{HeaderName, HeaderValue};
use tower_http::set_header::SetResponseHeaderLayer;

pub fn hsts_header() -> SetResponseHeaderLayer<HeaderValue> {
    SetResponseHeaderLayer::overriding(
        HeaderName::from_static("strict-transport-security"),
        // 2 years + include subdomains; add preload only after careful rollout.
        HeaderValue::from_static("max-age=63072000; includeSubDomains"),
    )
}
```

5.3 Request Size Limits and Body Validation

Request size limits protect memory and CPU. Enforce limits at both proxy and app.

Set a body size limit in the application

Use a default body limit suitable for your API. For JSON APIs, 1–2 MB is often enough; raise selectively for upload endpoints.

```rust
use axum::{routing::post, Router};
use axum::extract::DefaultBodyLimit;

pub fn router_with_limits() -> Router {
    Router::new()
        .route("/v1/users", post(crate::handlers::users::create_user))
        // 1 MiB default for the whole router tree
        .layer(DefaultBodyLimit::max(1024 * 1024))
}
```

Endpoint-specific larger limit

Apply higher limits only where needed.

```rust
use axum::{routing::post, Router};
use axum::extract::DefaultBodyLimit;

pub fn uploads_router() -> Router {
    Router::new()
        .route("/v1/upload", post(crate::handlers::upload::upload))
        .layer(DefaultBodyLimit::max(20 * 1024 * 1024)) // 20 MiB
}
```

Validate JSON bodies beyond syntax

JSON parsing validates syntax, not business rules. Validate fields explicitly and return structured errors.

```rust
use serde::Deserialize;

#[derive(Debug, Deserialize)]
pub struct CreateUserRequest {
    pub name: String,
    pub email: String,
}

pub fn validate_create_user(req: &CreateUserRequest) -> Result<(), &'static str> {
    let name = req.name.trim();
    if name.is_empty() { return Err("name is empty"); }
    if name.len() > 64 { return Err("name too long"); }

    let email = req.email.trim();
    if email.len() > 254 { return Err("email too long"); }
    if !email.contains('@') { return Err("email invalid"); }

    Ok(())
}
```

5.4 Protecting Against Common Attacks

This section focuses on practical, high-impact protections that fit API services.

5.4.1 1) Injection (SQL/Command/Template)

- Use parameterized queries; never build SQL by string concatenation.
- Avoid spawning shell commands with user input.

Example: parameterized query shape (conceptual)

```
// Pseudocode shape: always bind parameters, never format SQL with user input.
// let row = sqlx::query!("SELECT id, name FROM users WHERE email = $1",
↪  email).fetch_one(&pool).await?;
```

5.4.2 2) Authentication and session mistakes

- Prefer short-lived access tokens (and refresh tokens if needed).
- Validate signature, issuer, audience, and expiration.
- Enforce HTTPS-only cookies if using cookies.

5.4.3 3) CSRF

CSRF applies mainly to cookie-based authentication in browsers.

- If you use cookies for auth: require CSRF tokens or `SameSite=Strict/Lax` where appropriate.
- If you use bearer tokens in `Authorization`: CSRF is generally not applicable in the same way.

5.4.4 4) XSS and clickjacking

APIs serving JSON only: risk is lower. If you serve HTML:

- Use CSP and proper output escaping.
- Use `frame-ancestors 'none'` or a restrictive policy.

5.4.5 5) Request smuggling / header ambiguity

Mitigate primarily at the proxy:

- normalize and validate headers,
- reject ambiguous `Content-Length` / `Transfer-Encoding` combinations,
- keep proxy and app HTTP parsing standards-compliant.

5.4.6 6) Abuse: brute force, enumeration, overload

Combine layers:

- rate limiting and concurrency limiting,
- timeouts,
- consistent `404` and auth error responses to reduce oracle behavior,
- monitoring and alerting on suspicious patterns.

Example: combine limits and timeouts

```rust
use std::time::Duration;
use tower::ServiceBuilder;
use tower::limit::{ConcurrencyLimitLayer, RateLimitLayer};
use tower_http::timeout::TimeoutLayer;

pub fn abuse_protection_stack() -> ServiceBuilder<tower::layer::util::Identity> {
    ServiceBuilder::new()
        .layer(ConcurrencyLimitLayer::new(256))
        .layer(RateLimitLayer::new(100u64, Duration::from_secs(1)))
        .layer(TimeoutLayer::new(Duration::from_secs(3)))
}
```

5.5 Environment-Based Secret Management

Secrets must not be stored in source code, and typically not in config files committed to version control.

Baseline rules

- Secrets live in environment variables or a dedicated secret store.
- The application loads secrets at startup and fails fast if required secrets are missing.
- Never log secrets. Avoid printing full configuration.

Windows environment variable patterns

- Set for current PowerShell session: `$env:KEY="value"`
- Persist for the user (new terminals): `setx KEY value`

Example: strict secret loading (fail fast)

```
#[derive(Clone)]
pub struct Secrets {
    pub jwt_signing_key: String,
    pub db_password: String,
}

fn must_env(key: &str) -> String {
    std::env::var(key).unwrap_or_else(|_| panic!("missing required env var: {key}"))
}

pub fn load_secrets() -> Secrets {
    Secrets {
        jwt_signing_key: must_env("APP_JWT_SIGNING_KEY"),
        db_password: must_env("APP_DB_PASSWORD"),
    }
}
```

Example: .env for local Windows development only

Use `.env` locally, never commit secrets.

```
APP_JWT_SIGNING_KEY=dev-only-change-me
APP_DB_PASSWORD=dev-only-password
```

Integrating secrets into shared application state

```
use std::sync::Arc;

#[derive(Clone)]
pub struct AppState {
    pub secrets: Secrets,
}

pub fn build_state() -> Arc<AppState> {
    let _ = dotenvy::dotenv(); // ignore missing in production
    let secrets = load_secrets();
    Arc::new(AppState { secrets })
}
```

5.6 Windows Commands (PowerShell)

```
# Set secrets for current session
$env:APP_JWT_SIGNING_KEY = "dev-only-change-me"
```

```
$env:APP_DB_PASSWORD = "dev-only-password"

# Run
cargo run -p app

# Release build (Windows .exe)
cargo build -p app --release
```

Chapter outcomes

- You understand how TLS termination and forwarded headers affect trust boundaries.
- You can apply practical security headers (HSTS, CSP, and baselines) correctly.
- You can enforce request size limits and validate request bodies.
- You can mitigate common attacks via layered protections and strict input handling.
- You can manage secrets via environment variables in a Windows-native workflow.

Authentication and Authorization (JWT + Role-Based Access)

6.1 User Registration and Login Flow

A production authentication design must separate:

- **identity proof** (password verification),
- **session continuity** (refresh token rotation),
- **request authorization** (roles/permissions).

A clean and scalable flow:

- **Register**:
  - validate input
  - hash password with Argon2 using a per-user random salt
  - store user record (email unique) and password hash
- **Login**:
  - lookup user by email

  - verify password against stored Argon2 hash
  - issue short-lived **access token** + long-lived **refresh token**
  - store refresh token fingerprint (hashed) with rotation metadata

- **Refresh**:
  - verify refresh token
  - rotate: invalidate old refresh token, issue a new one
  - issue a new access token

- **Logout / Revoke**:
  - revoke refresh token family or a single token (server-side)

6.2 Secure Password Hashing with Argon2

Argon2 is designed for password hashing with memory-hardness. Use:

- per-password random salt,
- a server-side **pepper** stored in environment secrets,
- constant-time verification via the Argon2 library.

Dependencies (Cargo.toml)

```toml
[dependencies]
axum = "0.7"
tokio = { version = "1", features = ["rt-multi-thread", "macros"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
```

```toml
argon2 = "0.5"
password-hash = "0.5"
rand_core = "0.6"
subtle = "2"

jsonwebtoken = "9"
time = { version = "0.3", features = ["macros"] }
uuid = { version = "1", features = ["v4", "serde"] }

tower = "0.5"
tower-http = { version = "0.6", features = ["trace", "request-id"] }
```

Argon2 hashing and verification

```rust
use argon2::{
    password_hash::{PasswordHash, PasswordHasher, PasswordVerifier, SaltString},
    Argon2,
};
use rand_core::OsRng;

pub struct PasswordService {
    pepper: String,
}

impl PasswordService {
    pub fn new(pepper: String) -> Self { Self { pepper } }

    pub fn hash_password(&self, password: &str) -> Result<String, &'static str> {
        if password.len() < 12 { return Err("password too short"); }

        let salt = SaltString::generate(&mut OsRng);
        let argon2 = Argon2::default();
        // Pepper is appended (or mixed) on the server side.
        let secret = format!("{password}{}", self.pepper);

        let hash = argon2
            .hash_password(secret.as_bytes(), &salt)
            .map_err(|_| "hashing failed")?
            .to_string();

        Ok(hash)
    }

    pub fn verify_password(&self, password: &str, stored_hash: &str) -> Result<bool,
    ↪   &'static str> {
        let parsed = PasswordHash::new(stored_hash).map_err(|_| "invalid stored hash")?;
        let argon2 = Argon2::default();

        let secret = format!("{password}{}", self.pepper);

        Ok(argon2.verify_password(secret.as_bytes(), &parsed).is_ok())
    }
}
```

6.3 JWT Access and Refresh Tokens

6.3.1 Token design (practical and safe defaults)

- **Access token**: short-lived (e.g., 5–15 minutes), sent in `Authorization: Bearer ...`.
- **Refresh token**: longer-lived (e.g., days), used only at `/auth/refresh`.
- Include:

- `sub` (user id),
- `exp` (expiration),
- `iat` (issued-at),
- `jti` (token id for revocation/rotation),
- `typ` (access/refresh),
- roles/permissions for RBAC (access token).

JWT types and issuer

```rust
use serde::{Deserialize, Serialize};
use time::OffsetDateTime;
use uuid::Uuid;

#[derive(Debug, Clone, Serialize, Deserialize)]
pub enum TokenType {
    Access,
    Refresh,
}

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct Claims {
    pub sub: String,          // user id as string
    pub exp: i64,             // unix timestamp
    pub iat: i64,             // unix timestamp
    pub jti: String,          // unique token id
    pub typ: TokenType,       // access or refresh
    pub roles: Vec<String>,   // for RBAC (usually in access token)
    pub iss: String,          // issuer
    pub aud: String,          // audience
}
```

```
pub fn now_ts() -> i64 {
    OffsetDateTime::now_utc().unix_timestamp()
}

pub fn new_jti() -> String {
    Uuid::new_v4().to_string()
}
```

JWT service (HS256 example)

In many internal services HS256 is acceptable if the signing secret is well-protected and rotated. For multi-service ecosystems, consider asymmetric keys (RS256/ES256) and key IDs.

```
use jsonwebtoken::{DecodingKey, EncodingKey, Header, Validation, Algorithm, TokenData};
use jsonwebtoken::errors::Error as JwtError;

pub struct JwtService {
    issuer: String,
    audience: String,
    access_secret: Vec<u8>,
    refresh_secret: Vec<u8>,
    access_ttl_seconds: i64,
    refresh_ttl_seconds: i64,
}

impl JwtService {
    pub fn new(
        issuer: String,
        audience: String,
        access_secret: String,
        refresh_secret: String,
        access_ttl_seconds: i64,
        refresh_ttl_seconds: i64,
    ) -> Self {
        Self {
            issuer,
            audience,
            access_secret: access_secret.into_bytes(),
            refresh_secret: refresh_secret.into_bytes(),
            access_ttl_seconds,
            refresh_ttl_seconds,
        }
    }

    pub fn mint_access(&self, user_id: &str, roles: Vec<String>) -> Result<String,
     ↪   JwtError> {
        let iat = crate::auth::now_ts();
        let claims = crate::auth::Claims {
            sub: user_id.to_string(),
            iat,
            exp: iat + self.access_ttl_seconds,
            jti: crate::auth::new_jti(),
            typ: crate::auth::TokenType::Access,
            roles,
            iss: self.issuer.clone(),
            aud: self.audience.clone(),
        };

        jsonwebtoken::encode(
            &Header::new(Algorithm::HS256),
            &claims,
            &EncodingKey::from_secret(&self.access_secret),
        )
    }

    pub fn mint_refresh(&self, user_id: &str) -> Result<String, JwtError> {
        let iat = crate::auth::now_ts();
        let claims = crate::auth::Claims {
            sub: user_id.to_string(),
            iat,
            exp: iat + self.refresh_ttl_seconds,
            jti: crate::auth::new_jti(),
            typ: crate::auth::TokenType::Refresh,
            roles: Vec::new(), // keep refresh minimal
            iss: self.issuer.clone(),
            aud: self.audience.clone(),
        };

        jsonwebtoken::encode(
            &Header::new(Algorithm::HS256),
            &claims,
            &EncodingKey::from_secret(&self.refresh_secret),
        )
    }

    pub fn verify_access(&self, token: &str) -> Result<TokenData<crate::auth::Claims>,
    ↪  JwtError> {
        let mut v = Validation::new(Algorithm::HS256);
        v.set_issuer(&[self.issuer.clone()]);
        v.set_audience(&[self.audience.clone()]);
        jsonwebtoken::decode(
            token,
            &DecodingKey::from_secret(&self.access_secret),
            &v,
        )
    }

    pub fn verify_refresh(&self, token: &str) -> Result<TokenData<crate::auth::Claims>,
    ↪  JwtError> {
        let mut v = Validation::new(Algorithm::HS256);
        v.set_issuer(&[self.issuer.clone()]);
        v.set_audience(&[self.audience.clone()]);
        jsonwebtoken::decode(
            token,
            &DecodingKey::from_secret(&self.refresh_secret),
            &v,
        )
    }
}
```

6.4 Claims Extraction Middleware

The canonical API pattern:

- parse `Authorization: Bearer <token>`
- verify signature and standard claims
- inject verified `Claims` into request extensions
- handlers extract `Claims` when they need auth context

AppError and result type

```
use axum::{
    http::StatusCode,
    response::{IntoResponse, Response},
    Json,
};
use serde::Serialize;

pub type AppResult<T> = Result<T, AppError>;
```

```
#[derive(Debug)]
pub enum AppError {
    BadRequest(&'static str),
    Unauthorized(&'static str),
    Forbidden(&'static str),
    Conflict(&'static str),
    Internal(&'static str),
}

#[derive(Serialize)]
struct ErrorBody {
    code: u16,
    message: String,
}

impl IntoResponse for AppError {
    fn into_response(self) -> Response {
        let (status, msg) = match self {
            AppError::BadRequest(m) => (StatusCode::BAD_REQUEST, m),
            AppError::Unauthorized(m) => (StatusCode::UNAUTHORIZED, m),
            AppError::Forbidden(m) => (StatusCode::FORBIDDEN, m),
            AppError::Conflict(m) => (StatusCode::CONFLICT, m),
            AppError::Internal(m) => (StatusCode::INTERNAL_SERVER_ERROR, m),
        };
        (status, Json(ErrorBody { code: status.as_u16(), message: msg.to_string()
        ↪  })).into_response()
    }
}
```

Bearer parsing helper

```
pub fn bearer_token(auth_header: &str) -> Option<&str> {
    let trimmed = auth_header.trim();
    let (prefix, token) = trimmed.split_once(' ')?;
    if !prefix.eq_ignore_ascii_case("bearer") { return None; }
    let token = token.trim();
    if token.is_empty() { return None; }
    Some(token)
}
```

Auth middleware (Axum)

```
use axum::{http::Request, middleware::Next, response::Response};
use std::sync::Arc;

#[derive(Clone)]
pub struct AppState {
    pub jwt: Arc<crate::jwt::JwtService>,
}

pub async fn require_auth<B>(
    state: Arc<AppState>,
    mut req: Request<B>,
    next: Next<B>,
) -> Response {
    let token = match req.headers().get(axum::http::header::AUTHORIZATION).and_then(|v|
    ↪  v.to_str().ok()) {
        Some(v) => match crate::auth_mw::bearer_token(v) {
            Some(t) => t,
            None => return
            ↪  crate::error::AppError::Unauthorized("invalid authorization header").into_response
        },
        None => return
        ↪  crate::error::AppError::Unauthorized("missing authorization header").into_response(),
    };
    let data = match state.jwt.verify_access(token) {
        Ok(d) => d,
        Err(_) => return
        ↪  crate::error::AppError::Unauthorized("invalid or expired token").into_response(),
    };

    // Ensure token is access type.
    if !matches!(data.claims.typ, crate::auth::TokenType::Access) {
        return crate::error::AppError::Unauthorized("wrong token type").into_response();
    }

    req.extensions_mut().insert(data.claims);
    next.run(req).await
}
```

Extract claims in handlers

```
use axum::extract::Extension;

pub async fn me(Extension(claims): Extension<crate::auth::Claims>) -> String {
    format!("sub={} roles={:?}", claims.sub, claims.roles)
}
```

6.5 Role-Based Authorization (RBAC)

RBAC is best expressed as:

- roles in the access token (small list),
- a guard function to require a role,
- optional permission checks in domain logic.

Role guard utilities

```
pub fn has_role(claims: &crate::auth::Claims, role: &str) -> bool {
    claims.roles.iter().any(|r| r == role)
}

pub fn require_role(claims: &crate::auth::Claims, role: &'static str) -> Result<(),
↪   crate::error::AppError> {
    if has_role(claims, role) { Ok(()) }
    else { Err(crate::error::AppError::Forbidden("insufficient role")) }
}
```

Admin-only route handler

```
use axum::extract::Extension;
use crate::error::{AppError, AppResult};

pub async fn admin_metrics(Extension(claims): Extension<crate::auth::Claims>) ->
↪   AppResult<String> {
    crate::rbac::require_role(&claims, "admin")?;
    Ok("metrics: ok".to_string())
}
```

Routing: public vs protected vs admin

```
use axum::{routing::{get, post}, Router};
use std::sync::Arc;

pub fn router(state: Arc<crate::auth_mw::AppState>) -> Router {
    let public = Router::new()
        .route("/auth/register", post(crate::handlers::auth::register))
        .route("/auth/login", post(crate::handlers::auth::login))
        .route("/auth/refresh", post(crate::handlers::auth::refresh))
        .route("/health", get(crate::handlers::health::health));

    let protected = Router::new()
        .route("/me", get(crate::handlers::user::me))
        .layer(axum::middleware::from_fn({
            let st = state.clone();
            move |req, next| crate::auth_mw::require_auth(st.clone(), req, next)
        }));

    let admin = Router::new()
        .route("/admin/metrics", get(crate::handlers::admin::admin_metrics))
        .layer(axum::middleware::from_fn({
            let st = state.clone();
            move |req, next| crate::auth_mw::require_auth(st.clone(), req, next)
        }));

    Router::new()
        .merge(public)
        .merge(protected)
        .merge(admin)
}
```

6.6 End-to-End Example: Register, Login, Refresh

This example keeps storage in-memory to focus on security mechanics. In production, you must:

- store users in a database,
- store refresh token fingerprints (hashed) and implement rotation and revocation,
- enforce unique email, and audit suspicious login attempts.

In-memory repository (demo)

```rust
use std::{collections::HashMap, sync::Arc};
use tokio::sync::RwLock;

#[derive(Clone, Debug)]
pub struct UserRecord {
    pub id: String,
    pub email: String,
    pub password_hash: String,
    pub roles: Vec<String>,
}

#[derive(Clone, Default)]
pub struct UserRepo {
    inner: Arc<RwLock<HashMap<String, UserRecord>>>, // email -> record
}

impl UserRepo {
    pub async fn insert(&self, u: UserRecord) -> Result<(), &'static str> {
        let mut w = self.inner.write().await;
        if w.contains_key(&u.email) { return Err("email exists"); }
        w.insert(u.email.clone(), u);
        Ok(())
    }

    pub async fn find_by_email(&self, email: &str) -> Option<UserRecord> {
        let r = self.inner.read().await;
        r.get(email).cloned()
    }
}
```

Auth state

```rust
#[derive(Clone)]
pub struct AuthState {
    pub users: crate::repo::UserRepo,
    pub passwords: crate::passwords::PasswordService,
    pub jwt: std::sync::Arc<crate::jwt::JwtService>,
}
```

Auth handlers

```rust
use axum::{extract::State, Json};
use serde::{Deserialize, Serialize};
use uuid::Uuid;
use crate::error::{AppError, AppResult};

#[derive(Debug, Deserialize)]
pub struct RegisterReq {
    pub email: String,
    pub password: String,
}

#[derive(Debug, Serialize)]
pub struct RegisterResp {
    pub user_id: String,
}

pub async fn register(
    State(st): State<std::sync::Arc<crate::state::AuthState>>,
    Json(req): Json<RegisterReq>,
) -> AppResult<Json<RegisterResp>> {
    let email = req.email.trim().to_lowercase();
    if email.is_empty() { return Err(AppError::BadRequest("email empty")); }
    if !email.contains('@') { return Err(AppError::BadRequest("email invalid")); }

    let hash = st.passwords.hash_password(&req.password)
        .map_err(|_| AppError::BadRequest("password rejected"))?;

    let user_id = Uuid::new_v4().to_string();

    let rec = crate::repo::UserRecord {
        id: user_id.clone(),
        email: email.clone(),
        password_hash: hash,
        roles: vec!["user".to_string()],
    };

    st.users.insert(rec).await.map_err(|_| AppError::Conflict("email already exists"))?;
    Ok(Json(RegisterResp { user_id }))
}

#[derive(Debug, Deserialize)]
pub struct LoginReq {
    pub email: String,
    pub password: String,
}

#[derive(Debug, Serialize)]
pub struct TokenPair {
    pub access_token: String,
    pub refresh_token: String,
}

pub async fn login(
    State(st): State<std::sync::Arc<crate::state::AuthState>>,
    Json(req): Json<LoginReq>,
) -> AppResult<Json<TokenPair>> {
    let email = req.email.trim().to_lowercase();
    let user = st.users.find_by_email(&email).await
        .ok_or(AppError::Unauthorized("invalid credentials"))?;

    let ok = st.passwords.verify_password(&req.password, &user.password_hash)
        .map_err(|_| AppError::Unauthorized("invalid credentials"))?;

    if !ok { return Err(AppError::Unauthorized("invalid credentials")); }

    let access = st.jwt.mint_access(&user.id, user.roles.clone())
        .map_err(|_| AppError::Internal("token mint failed"))?;

    let refresh = st.jwt.mint_refresh(&user.id)
        .map_err(|_| AppError::Internal("token mint failed"))?;

    Ok(Json(TokenPair { access_token: access, refresh_token: refresh }))
}

#[derive(Debug, Deserialize)]
pub struct RefreshReq {
    pub refresh_token: String,
}

pub async fn refresh(
    State(st): State<std::sync::Arc<crate::state::AuthState>>,
    Json(req): Json<RefreshReq>,
) -> AppResult<Json<TokenPair>> {
    let data = st.jwt.verify_refresh(&req.refresh_token)
        .map_err(|_| AppError::Unauthorized("invalid refresh token"))?;

    if !matches!(data.claims.typ, crate::auth::TokenType::Refresh) {
        return Err(AppError::Unauthorized("wrong token type"));
    }

    // Production requirement: verify refresh token family, rotation, revocation
    ↪  server-side.

    let user_id = data.claims.sub;
    let access = st.jwt.mint_access(&user_id, vec!["user".to_string()])
        .map_err(|_| AppError::Internal("token mint failed"))?;

    let refresh = st.jwt.mint_refresh(&user_id)
        .map_err(|_| AppError::Internal("token mint failed"))?;

    Ok(Json(TokenPair { access_token: access, refresh_token: refresh }))
}
```

Wire state (Windows secrets via environment variables)

```
pub fn must_env(key: &str) -> String {
    std::env::var(key).unwrap_or_else(|_| panic!("missing env var: {key}"))
}

pub fn build_auth_state() -> std::sync::Arc<crate::state::AuthState> {
    let _ = dotenvy::dotenv(); // ok if missing

    let pepper = must_env("APP_PASSWORD_PEPPER");

    let issuer = must_env("APP_JWT_ISSUER");
    let audience = must_env("APP_JWT_AUDIENCE");
    let access_secret = must_env("APP_JWT_ACCESS_SECRET");
    let refresh_secret = must_env("APP_JWT_REFRESH_SECRET");

    let passwords = crate::passwords::PasswordService::new(pepper);
    let jwt = std::sync::Arc::new(crate::jwt::JwtService::new(
        issuer,
        audience,
        access_secret,
        refresh_secret,
        10 * 60,          // access 10 minutes
        7 * 24 * 3600,    // refresh 7 days
    ));

    std::sync::Arc::new(crate::state::AuthState {
        users: crate::repo::UserRepo::default(),
        passwords,
        jwt,
    })
}
```

6.7 Windows Commands (PowerShell)

```
# Required secrets for local development (example values only)
$env:APP_PASSWORD_PEPPER     = "dev-pepper-change-me"
$env:APP_JWT_ISSUER          = "rust-backend-engineering"
$env:APP_JWT_AUDIENCE        = "rust-backend-engineering-api"
$env:APP_JWT_ACCESS_SECRET   = "dev-access-secret-change-me-very-long"
$env:APP_JWT_REFRESH_SECRET  = "dev-refresh-secret-change-me-very-long"

# Run
cargo run -p app

# Release build produces a Windows .exe
cargo build -p app --release

# Quality gates
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all
```

Chapter outcomes

- You can implement registration, login, and refresh flows with clear security boundaries.
- You can hash and verify passwords using Argon2 with salt and server-side pepper.
- You can mint and verify access/refresh JWTs with explicit claim design.
- You can extract claims via middleware and enforce RBAC with reusable guards.

Database Integration with SQLx

7.1 Connection Pooling

SQLx provides async, runtime-agnostic, compile-time checked SQL with robust connection pooling. In production backends, **never open a database connection per request**. Always use a pool.

Dependencies (Cargo.toml, Windows + Tokio)

```
[dependencies]
tokio = { version = "1", features = ["rt-multi-thread", "macros"] }
sqlx = { version = "0.7", features = [
    "runtime-tokio-rustls",
    "postgres",
    "uuid",
    "time",
    "macros"
] }
uuid = { version = "1", features = ["v4", "serde"] }
time = { version = "0.3", features = ["macros", "serde"] }
serde = { version = "1", features = ["derive"] }
```

Creating a connection pool

```
use sqlx::{Pool, Postgres, postgres::PgPoolOptions};
use std::time::Duration;

pub async fn create_pool(database_url: &str) -> Result<Pool<Postgres>, sqlx::Error> {
    PgPoolOptions::new()
        .max_connections(20)
        .min_connections(5)
        .acquire_timeout(Duration::from_secs(5))
        .idle_timeout(Duration::from_secs(600))
        .connect(database_url)
        .await
}
```

Store pool in application state

```
use std::sync::Arc;
use sqlx::{Pool, Postgres};

#[derive(Clone)]
pub struct AppState {
    pub db: Pool<Postgres>,
}

pub fn new_state(pool: Pool<Postgres>) -> Arc<AppState> {
    Arc::new(AppState { db: pool })
}
```

7.2 Repository Pattern (Lightweight Design)

Avoid heavy ORM-style abstractions. A **lightweight repository**:

- isolates SQL from handlers,
- accepts a `&Pool<Postgres>` or `&mut Transaction`,
- returns domain types or DTOs,
- does not embed business logic.

Domain model

```
use serde::{Serialize, Deserialize};
use uuid::Uuid;
use time::OffsetDateTime;

#[derive(Debug, Clone, Serialize, Deserialize)]
pub struct User {
    pub id: Uuid,
    pub email: String,
    pub created_at: OffsetDateTime,
}
```

User repository

```
use sqlx::{Pool, Postgres};
use uuid::Uuid;
use time::OffsetDateTime;

pub struct UserRepository;

impl UserRepository {

    pub async fn insert(
        pool: &Pool<Postgres>,
        email: &str,
    ) -> Result<User, sqlx::Error> {

        let rec = sqlx::query_as!(
            User,
            r#"
            INSERT INTO users (id, email, created_at)
            VALUES ($1, $2, now())
            RETURNING id, email, created_at
            "#,
            Uuid::new_v4(),
            email
        )
        .fetch_one(pool)
        .await?;

        Ok(rec)
    }

    pub async fn find_by_id(
        pool: &Pool<Postgres>,
        id: Uuid,
    ) -> Result<Option<User>, sqlx::Error> {

        let rec = sqlx::query_as!(
            User,
            r#"
            SELECT id, email, created_at
            FROM users
            WHERE id = $1
            "#,
            id
        )
```

```
        .fetch_optional(pool)
        .await?;

        Ok(rec)
    }

    pub async fn delete(
        pool: &Pool<Postgres>,
        id: Uuid,
    ) -> Result<u64, sqlx::Error> {

        let result = sqlx::query!(
            "DELETE FROM users WHERE id = $1",
            id
        )
        .execute(pool)
        .await?;

        Ok(result.rows_affected())
    }
}
```

7.3 Database Migrations

SQLx provides first-class migration support via `sqlx-cli`.

Install CLI on Windows (PowerShell)

```
cargo install sqlx-cli --no-default-features --features postgres,rustls
```

Initialize migrations directory

```
sqlx migrate add create_users_table
```

Example migration file

```sql
-- migrations/20240101000000_create_users_table.sql

CREATE TABLE IF NOT EXISTS users (
    id UUID PRIMARY KEY,
    email TEXT NOT NULL UNIQUE,
    created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

Run migrations

```
$env:DATABASE_URL="postgres://postgres:password@localhost:5432/mydb"
sqlx migrate run
```

Embedded migrations (optional production pattern)

```rust
use sqlx::migrate::Migrator;

static MIGRATOR: Migrator = sqlx::migrate!();

pub async fn run_migrations(pool: &sqlx::Pool<sqlx::Postgres>) -> Result<(), sqlx::Error> {
    MIGRATOR.run(pool).await
}
```

7.4 CRUD Operations

7.4.1 Create

```
pub async fn create_user(
    pool: &sqlx::Pool<sqlx::Postgres>,
    email: String,
) -> Result<User, sqlx::Error> {
    UserRepository::insert(pool, &email).await
}
```

7.4.2 Read

```
pub async fn get_user(
    pool: &sqlx::Pool<sqlx::Postgres>,
    id: uuid::Uuid,
) -> Result<Option<User>, sqlx::Error> {
    UserRepository::find_by_id(pool, id).await
}
```

7.4.3 Update

```
pub async fn update_user_email(
    pool: &sqlx::Pool<sqlx::Postgres>,
    id: uuid::Uuid,
    email: String,
) -> Result<Option<User>, sqlx::Error> {

    let rec = sqlx::query_as!(
        User,
        r#"
        UPDATE users
        SET email = $1
        WHERE id = $2
        RETURNING id, email, created_at
        "#,
        email,
        id
    )
    .fetch_optional(pool)
    .await?;

    Ok(rec)
}
```

7.4.4 Delete

```
pub async fn delete_user(
    pool: &sqlx::Pool<sqlx::Postgres>,
    id: uuid::Uuid,
) -> Result<bool, sqlx::Error> {
    let affected = UserRepository::delete(pool, id).await?;
    Ok(affected > 0)
}
```

7.5 Safe Error Propagation

Never expose raw database errors directly to clients. Convert `sqlx::Error` into application-level errors.

Application error type

```
use axum::{http::StatusCode, response::{IntoResponse, Response}, Json};
use serde::Serialize;
```

```rust
#[derive(Debug)]
pub enum AppError {
    NotFound,
    Conflict(&'static str),
    Internal(&'static str),
}

#[derive(Serialize)]
struct ErrorBody {
    code: u16,
    message: String,
}

impl IntoResponse for AppError {
    fn into_response(self) -> Response {
        let (status, message) = match self {
            AppError::NotFound => (StatusCode::NOT_FOUND, "not found"),
            AppError::Conflict(m) => (StatusCode::CONFLICT, m),
            AppError::Internal(m) => (StatusCode::INTERNAL_SERVER_ERROR, m),
        };

        (status, Json(ErrorBody {
            code: status.as_u16(),
            message: message.to_string(),
        })).into_response()
    }
}
```

Mapping SQLx errors safely

```rust
impl From<sqlx::Error> for AppError {
    fn from(e: sqlx::Error) -> Self {
        match &e {
            sqlx::Error::RowNotFound => AppError::NotFound,
            sqlx::Error::Database(db_err) => {
                // Unique constraint example (PostgreSQL code 23505)
                if db_err.code().as_deref() == Some("23505") {
                    AppError::Conflict("duplicate value")
                } else {
                    AppError::Internal("database error")
                }
            }
            _ => AppError::Internal("database error"),
        }
    }
}
```

Handler example with safe propagation

```
use axum::{extract::{State, Path}, Json};
use std::sync::Arc;
use uuid::Uuid;

pub async fn get_user_handler(
    State(state): State<Arc<AppState>>,
    Path(id): Path<Uuid>,
) -> Result<Json<User>, AppError> {

    let user = UserRepository::find_by_id(&state.db, id)
        .await?
        .ok_or(AppError::NotFound)?;

    Ok(Json(user))
}
```

7.6 Windows Development Commands

```
# Set database URL
$env:DATABASE_URL="postgres://postgres:password@localhost:5432/mydb"

# Run migrations
sqlx migrate run

# Run application
cargo run -p app

# Release build
cargo build -p app --release

# Lint and test
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all
```

Chapter Outcomes

- You implemented robust connection pooling with SQLx.
- You applied a lightweight repository pattern that isolates SQL cleanly.
- You managed migrations using SQLx CLI and embedded migration support.
- You implemented full CRUD with compile-time checked queries.
- You safely mapped database errors into structured application responses.

Observability and Monitoring

8.1 Structured Logging with Tracing

In production, logs must be:

- **structured** (fields, not just strings),
- **consistent** (same keys across services),
- **correlated** (request IDs / trace IDs),
- **safe** (no secrets, no PII leakage).

Rusts standard approach is the `tracing` ecosystem:

- `tracing`: API for events and spans
- `tracing-subscriber`: formatting, filtering, env-based configuration
- `tower-http` tracing layer for HTTP request lifecycle

Windows logging setup (infrastructure crate)

```
//// crates/infrastructure/src/logging.rs
use tracing_subscriber::{fmt, EnvFilter};
```

```
pub fn init_logging(default_level: &str) {
    // Prefer RUST_LOG if set (PowerShell: $env:RUST_LOG="info")
    let filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new(default_level));

    fmt()
        .with_env_filter(filter)
        .with_target(false)
        .with_thread_names(true)
        .with_thread_ids(true)
        .compact()
        .init();
}
```

Use consistent fields

Choose stable keys so dashboards and alerts remain usable after refactors:

- `service`, `env`, `request_id`
- `http.method`, `http.route`, `http.status`
- `latency_ms`, `error`

Event examples

```
use tracing::{info, warn, error};

pub fn log_examples() {
    info!(service="rbe", env="local", "service starting");
    warn!(component="db", "pool is warming up");
    error!(component="auth", user_id=%42, "invalid token signature");
}
```

8.2 Request-Scoped Spans

A **span** represents an operation with context. Request-scoped spans:

- attach request metadata once,
- automatically apply that context to all nested logs,
- make debugging multi-step flows practical.

HTTP tracing with `tower-http`

```rust
use tower_http::trace::TraceLayer;

pub fn http_trace_layer() ->
↪  TraceLayer<tower_http::classify::SharedClassifier<tower_http::classify::ServerErrorsAsFailures
↪  {
    TraceLayer::new_for_http()
}
```

Attach request ID early

```rust
use axum::http::HeaderName;
use tower::ServiceBuilder;
use tower_http::request_id::{MakeRequestUuid, SetRequestIdLayer, PropagateRequestIdLayer};

pub fn request_id_stack() -> ServiceBuilder<tower::layer::util::Identity> {
    let name = HeaderName::from_static("x-request-id");
    ServiceBuilder::new()
        .layer(SetRequestIdLayer::new(name.clone(), MakeRequestUuid))
        .layer(PropagateRequestIdLayer::new(name))
}
```

Custom span fields (route + request id)

When you want stricter control, create your own span.

```rust
use axum::{http::Request, middleware::Next, response::Response};
use tracing::{info_span, Instrument};

pub async fn span_middleware<B>(req: Request<B>, next: Next<B>) -> Response {
    let method = req.method().to_string();
    let path = req.uri().path().to_string();

    let request_id = req.headers()
        .get("x-request-id")
        .and_then(|v| v.to_str().ok())
        .unwrap_or("-")
        .to_string();

    let span = info_span!(
        "http.request",
        http_method = %method,
        http_path = %path,
        request_id = %request_id
    );

    next.run(req).instrument(span).await
}
```

Handler logs inherit the span automatically

```rust
use tracing::info;

pub async fn handler() -> &'static str {
    info!("handler entered");
    "ok"
}
```

8.3 Health and Readiness Endpoints

8.3.1 Definitions (practical operations model)

- **Liveness** (`/health`): process is running and event loop is responsive.
- **Readiness** (`/ready`): service can accept traffic (DB reachable, migrations applied, required config loaded).

Implementation approach

- `/health`: never call external dependencies; must be fast and stable.
- `/ready`: perform minimal dependency checks with tight timeouts.

Health handler

```rust
use axum::response::IntoResponse;

pub async fn health() -> impl IntoResponse {
    "ok"
}
```

Readiness with DB check (SQLx)

```rust
use axum::{extract::State, http::StatusCode, response::IntoResponse};
use std::{sync::Arc, time::Duration};

```

```rust
#[derive(Clone)]
pub struct AppState {
    pub db: sqlx::Pool<sqlx::Postgres>,
}

pub async fn ready(State(st): State<Arc<AppState>>) -> impl IntoResponse {
    let op = async {
        sqlx::query_scalar::<_, i64>("SELECT 1")
            .fetch_one(&st.db)
            .await
            .map(|_| ())
    };

    match tokio::time::timeout(Duration::from_millis(200), op).await {
        Ok(Ok(())) => (StatusCode::OK, "ready"),
        _ => (StatusCode::SERVICE_UNAVAILABLE, "not ready"),
    }
}
```

Routes

```rust
use axum::{routing::get, Router};

pub fn observability_routes() -> Router {
    Router::new()
        .route("/health", get(crate::handlers::health::health))
        .route("/ready", get(crate::handlers::health::ready))
}
```

8.4 Basic Metrics Strategy

8.4.1 What to measure first

Start with a minimal set that supports SLOs and incident response:

- request rate (RPS) by route/method
- latency (p50/p95/p99) by route
- error rate by status class (2xx/4xx/5xx)
- saturation signals (in-flight requests, DB pool usage)

8.4.2 A simple, dependency-light metrics approach

If you do not want a full metrics stack immediately:

- emit **structured log events** for key metrics
- aggregate them in your log pipeline (ELK/OpenSearch/Cloud logs)

Example: emit request metric events from middleware

```rust
use axum::{http::Request, middleware::Next, response::Response};
use std::time::Instant;
use tracing::info;

pub async fn metrics_middleware<B>(req: Request<B>, next: Next<B>) -> Response {
    let start = Instant::now();
    let method = req.method().to_string();
    let path = req.uri().path().to_string();
    let res = next.run(req).await;
    let status = res.status().as_u16();
    let latency_ms = start.elapsed().as_millis() as u64;

    info!(
        event="http_metric",
        http_method=%method,
        http_path=%path,
        http_status=status,
        latency_ms=latency_ms
    );

    res
}
```

8.4.3 Prometheus-style endpoint (minimal example)

If you do want a metrics endpoint quickly, keep it simple:

- expose a `/metrics` endpoint
- return text format from a shared registry

Below is a minimal approach using in-memory counters without external crates (suitable for learning). In production, you typically use a metrics crate and exporter.

Minimal counters (learning-grade)

```
use std::sync::atomic::{AtomicU64, Ordering};
use std::sync::Arc;

#[derive(Default)]
pub struct Metrics {
    pub requests_total: AtomicU64,
    pub errors_total: AtomicU64,
}

impl Metrics {
    pub fn inc_requests(&self) { self.requests_total.fetch_add(1, Ordering::Relaxed); }
    pub fn inc_errors(&self) { self.errors_total.fetch_add(1, Ordering::Relaxed); }
}

#[derive(Clone)]
pub struct MetricsState {
    pub metrics: Arc<Metrics>,
}
```

Metrics middleware increments counters

```
use axum::{http::Request, middleware::Next, response::Response};
use std::sync::Arc;

pub async fn counters_middleware<B>(
    st: Arc<crate::metrics::MetricsState>,
    req: Request<B>,
    next: Next<B>,
) -> Response {
    st.metrics.inc_requests();
    let res = next.run(req).await;
    if res.status().as_u16() >= 500 {
        st.metrics.inc_errors();
    }
    res
}
```

Expose `/metrics` as text

```
use axum::{extract::State, response::IntoResponse};
use std::sync::Arc;

pub async fn metrics(State(st): State<Arc<crate::metrics::MetricsState>>) -> impl
↪  IntoResponse {
    let r = st.metrics.requests_total.load(std::sync::atomic::Ordering::Relaxed);
    let e = st.metrics.errors_total.load(std::sync::atomic::Ordering::Relaxed);

    // Prometheus text format style (very small subset)
    format!(

        ↪  "# TYPE requests_total counter\nrequests_total {}\n# TYPE errors_total counter\nerrors_
        r, e
    )
}
```

Wire observability middleware and routes (Windows-ready)

```
use axum::{routing::get, Router};
use std::sync::Arc;

pub fn build_app_with_observability(base: Router) -> Router {
    let metrics_state = Arc::new(crate::metrics::MetricsState {
        metrics: Arc::new(crate::metrics::Metrics::default()),
    });

    let obs_routes = Router::new()
        .route("/metrics", get(crate::handlers::metrics::metrics))
        .with_state(metrics_state.clone());

    let with_mw = base
```

```
        .layer(axum::middleware::from_fn(crate::middleware::metrics_middleware));

    with_mw.merge(obs_routes)
}
```

8.5 Windows Commands (PowerShell)

```
# Set log filter for current session
$env:RUST_LOG = "info"

# Run
cargo run -p app

# Release build
cargo build -p app --release

# Check endpoints locally (PowerShell)
# Invoke-WebRequest http://127.0.0.1:8080/health
# Invoke-WebRequest http://127.0.0.1:8080/ready
# Invoke-WebRequest http://127.0.0.1:8080/metrics
```

Chapter outcomes

- You initialized structured logging with `tracing` and environment-driven filters on Windows.
- You created request-scoped spans so all logs correlate to a request automatically.
- You implemented liveness and readiness endpoints with correct operational semantics.
- You adopted a minimal metrics strategy that scales from log-based metrics to dedicated metrics exporters.

Testing Strategy for Production Backends

9.1 Unit Testing Domain Logic

The fastest and most valuable tests live in the `domain` crate:

- no HTTP, no database, no network
- deterministic inputs and outputs
- types enforce invariants

Example domain logic (pure)

```rust
//// crates/domain/src/services/user_service.rs
#[derive(Debug, Clone, PartialEq, Eq)]
pub struct NewUser {
    pub name: String,
    pub email: String,
}

pub fn validate_new_user(name: &str, email: &str) -> Result<NewUser, &'static str> {
    let name = name.trim();
    let email = email.trim().to_lowercase();
    if name.is_empty() { return Err("name empty"); }
    if name.len() > 64 { return Err("name too long"); }

    if email.is_empty() { return Err("email empty"); }
    if email.len() > 254 { return Err("email too long"); }
    if !email.contains('@') { return Err("email invalid"); }

    Ok(NewUser { name: name.to_string(), email })
}
```

Unit tests (fast, no runtime)

```
//// crates/domain/src/services/user_service.rs (tests)
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn validates_ok() {
        let u = validate_new_user("Ayman", "Ayman@Example.com").unwrap();
        assert_eq!(u.name, "Ayman");
        assert_eq!(u.email, "ayman@example.com");
    }

    #[test]
    fn rejects_empty_name() {
        assert!(validate_new_user("   ", "x@y.com").is_err());
    }

    #[test]
    fn rejects_invalid_email() {
        assert!(validate_new_user("A", "not-an-email").is_err());
    }
}
```

Windows commands

```
# Run unit tests only (all crates)
cargo test

# Or domain crate only
cargo test -p domain
```

9.2 Integration Testing Routes

Integration tests should validate:

- routing and extraction (path/query/json)
- status codes and error mapping
- middleware behavior (auth, timeouts, headers)

A practical pattern for Axum:

- expose a function that builds your `Router` (`api::routes::router(...)`)
- in tests, call the router directly using `tower::ServiceExt::oneshot`

A tiny router with handlers (API crate)

```
//// crates/api/src/routes.rs (excerpt)
use axum::{routing::{get, post}, Router};

pub fn router() -> Router {
    Router::new()
```

```
        .route("/health", get(crate::handlers::health::health))
        .route("/echo", post(crate::handlers::echo::echo_json))
}
```

```
//// crates/api/src/handlers/echo.rs
use axum::Json;
use serde::{Deserialize, Serialize};

#[derive(Debug, Deserialize)]
pub struct EchoReq {
    pub msg: String,
}

#[derive(Debug, Serialize)]
pub struct EchoResp {
    pub msg: String,
}

pub async fn echo_json(Json(req): Json<EchoReq>) -> Json<EchoResp> {
    Json(EchoResp { msg: req.msg })
}
```

Integration tests using oneshot

```
//// crates/api/tests/routes.rs
use axum::{body::Body, http::{Request, StatusCode}};
use tower::ServiceExt; // oneshot
use serde_json::json;

#[tokio::test]
async fn health_is_ok() {
    let app = api::routes::router();
    let res = app.oneshot(
        Request::builder()
            .method("GET")
            .uri("/health")
            .body(Body::empty())
            .unwrap()
    ).await.unwrap();

    assert_eq!(res.status(), StatusCode::OK);
}

#[tokio::test]
async fn echo_roundtrip() {
    let app = api::routes::router();

    let body = Body::from(json!({ "msg": "hello" }).to_string());

    let res = app.oneshot(
        Request::builder()
            .method("POST")
            .uri("/echo")
            .header("content-type", "application/json")
            .body(body)
            .unwrap()
    ).await.unwrap();

    assert_eq!(res.status(), StatusCode::OK);
}
```

Windows commands

```
# Run integration tests (all)
cargo test
```

```
# Run only API tests
cargo test -p api
```

9.3 Mocking and In-Memory Databases

Two common strategies:

- **mock repositories** (fast, deterministic)
- **real DB in a disposable environment** (more realistic, slower)

For production-grade backends, mock at boundaries but still keep a smaller set of tests that hit a real database.

9.3.1 Mocking with a repository trait

Define a small trait that represents what the domain/API needs, not the whole database.

Trait

```
//// crates/domain/src/ports/user_repo.rs
use async_trait::async_trait;
use uuid::Uuid;

#[derive(Debug, Clone)]
pub struct UserRow {
    pub id: Uuid,
    pub email: String,
}

#[async_trait]
pub trait UserRepo: Send + Sync {
    async fn find_by_id(&self, id: Uuid) -> Result<Option<UserRow>, &'static str>;
    async fn insert(&self, email: &str) -> Result<UserRow, &'static str>;
}
```

In-memory mock implementation (good for tests)

```
use std::{collections::HashMap, sync::Arc};
use tokio::sync::RwLock;
use uuid::Uuid;

#[derive(Clone, Default)]
pub struct InMemoryUserRepo {
    inner: Arc<RwLock<HashMap<Uuid, String>>>,
}

#[async_trait::async_trait]
impl crate::ports::user_repo::UserRepo for InMemoryUserRepo {
    async fn find_by_id(&self, id: Uuid) ->
    ↪  Result<Option<crate::ports::user_repo::UserRow>, &'static str> {
        let r = self.inner.read().await;
        Ok(r.get(&id).map(|email| crate::ports::user_repo::UserRow { id, email:
        ↪  email.clone() }))
    }

    async fn insert(&self, email: &str) -> Result<crate::ports::user_repo::UserRow,
    ↪  &'static str> {
        let id = Uuid::new_v4();
        let mut w = self.inner.write().await;
        w.insert(id, email.to_string());
        Ok(crate::ports::user_repo::UserRow { id, email: email.to_string() })
    }
}
```

Use mock repo in a service

```
use uuid::Uuid;
use crate::ports::user_repo::{UserRepo, UserRow};

pub struct UserService<R: UserRepo> {
    repo: R,
}

impl<R: UserRepo> UserService<R> {
    pub fn new(repo: R) -> Self { Self { repo } }

    pub async fn register(&self, email: &str) -> Result<UserRow, &'static str> {
        if !email.contains('@') { return Err("email invalid"); }
        self.repo.insert(email).await
    }

    pub async fn get(&self, id: Uuid) -> Result<Option<UserRow>, &'static str> {
        self.repo.find_by_id(id).await
    }
}
```

Service tests with mock repo

```
#[tokio::test]
async fn service_registers_user() {
    let repo = InMemoryUserRepo::default();
    let svc = UserService::new(repo);

    let u = svc.register("ayman@example.com").await.unwrap();
    let fetched = svc.get(u.id).await.unwrap().unwrap();
    assert_eq!(fetched.email, "ayman@example.com");
}
```

9.3.2 In-memory databases

For relational behavior, prefer a real DB in tests. If you want an in-memory approach, SQLite is often used for fast local runs, but it may differ from PostgreSQL semantics. Treat it as a **developer-speed test**, not a full production substitute.

SQLx + SQLite in-memory (fast test layer)

```
# Example if you add SQLite for tests
# sqlx = { version = "0.7", features = ["runtime-tokio-rustls", "sqlite", "macros"] }
```

```
use sqlx::{Pool, Sqlite, sqlite::SqlitePoolOptions};

pub async fn sqlite_mem_pool() -> Result<Pool<Sqlite>, sqlx::Error> {
    SqlitePoolOptions::new()
        .max_connections(1)
        .connect("sqlite::memory:")
        .await
}
```

9.4 Testing Authentication Flows

Auth testing must confirm:

- password hashing and verification correctness
- JWT minting, expiry, and claim validation
- middleware rejects missing/invalid tokens
- RBAC denies unauthorized roles

9.4.1 Password hashing tests (Argon2)

```rust
#[test]
fn password_hash_roundtrip() {
    let svc = crate::passwords::PasswordService::new("pepper".to_string());
    let h = svc.hash_password("very-strong-password").unwrap();
    assert!(svc.verify_password("very-strong-password", &h).unwrap());
    assert!(!svc.verify_password("wrong", &h).unwrap());
}
```

9.4.2 JWT tests (mint + verify)

```rust
#[test]
fn jwt_access_roundtrip() {
    let jwt = crate::jwt::JwtService::new(
        "issuer".to_string(),
        "aud".to_string(),
        "access-secret-very-long".to_string(),
        "refresh-secret-very-long".to_string(),
        600,
        7 * 24 * 3600,
    );

    let token = jwt.mint_access("u1", vec!["user".to_string()]).unwrap();
    let data = jwt.verify_access(&token).unwrap();
    assert_eq!(data.claims.sub, "u1");
    assert!(matches!(data.claims.typ, crate::auth::TokenType::Access));
}
```

9.4.3 Middleware rejection test (Axum + oneshot)

```rust
use axum::{body::Body, http::{Request, StatusCode}};
use tower::ServiceExt;
```

```rust
#[tokio::test]
async fn protected_route_requires_token() {
    let state = std::sync::Arc::new(crate::auth_mw::AppState {
        jwt: std::sync::Arc::new(crate::jwt::JwtService::new(
            "issuer".to_string(),
            "aud".to_string(),
            "access-secret-very-long".to_string(),
            "refresh-secret-very-long".to_string(),
            600,
            7 * 24 * 3600,
        )),
    });

    let app = axum::Router::new()
        .route("/me", axum::routing::get(|| async { "ok" }))
        .layer(axum::middleware::from_fn({
            let st = state.clone();
            move |req, next| crate::auth_mw::require_auth(st.clone(), req, next)
        }));

    let res = app.oneshot(
        Request::builder()
            .method("GET")
            .uri("/me")
            .body(Body::empty())
            .unwrap()
    ).await.unwrap();

    assert_eq!(res.status(), StatusCode::UNAUTHORIZED);
}
```

9.4.4 RBAC test (role guard)

```
#[test]
fn rbac_denies_missing_role() {
    let c = crate::auth::Claims {
        sub: "u1".to_string(),
        exp: 9999999999,
        iat: 1,
        jti: "x".to_string(),
        typ: crate::auth::TokenType::Access,
        roles: vec!["user".to_string()],
        iss: "issuer".to_string(),
        aud: "aud".to_string(),
    };

    let r = crate::rbac::require_role(&c, "admin");
    assert!(r.is_err());
}
```

9.5 Windows Test Commands (PowerShell)

```
# Run all tests
cargo test

# Run a specific crate
cargo test -p domain
cargo test -p api

# Run one test by name (substring)
cargo test protected_route_requires_token

# Show logs during tests
```

```
$env:RUST_LOG="info"
cargo test -- --nocapture
```

Chapter outcomes

- You separated fast unit tests (domain) from integration tests (routing/middleware).
- You used `oneshot` to test Axum routes without a real network port.
- You applied lightweight mocks and in-memory stores for deterministic tests.
- You validated authentication and authorization behavior with targeted tests.

Deployment and Production Readiness

10.1 Building Optimized Release Binaries

Production Rust services should be built with deterministic, optimized release settings and a reproducible toolchain.

Toolchain pinning (repository root)

```toml
# rust-toolchain.toml
[toolchain]
channel = "stable"
components = ["rustfmt", "clippy", "rust-src"]
```

Release profile (workspace root Cargo.toml)

These settings are widely used for server binaries (optimize for runtime, smaller output).

```toml
# Cargo.toml (workspace root)

[profile.release]
lto = true
codegen-units = 1
panic = "abort"
strip = "symbols"
```

Windows build commands (PowerShell)

```
# Format + lint + test gates
cargo fmt --all
cargo clippy --all-targets --all-features -- -D warnings
cargo test --all

# Release build (creates .exe on Windows)
cargo build -p app --release

# Run the built executable (example)
.\target\release\app.exe
```

Version stamping (build-time metadata)

A practical pattern is to expose `-version` or a `/version` endpoint using build-time env.

```
//// crates/app/src/build_info.rs
pub fn version() -> &'static str {
    env!("CARGO_PKG_VERSION")
}

pub fn git_sha() -> &'static str {
    option_env!("GIT_SHA").unwrap_or("unknown")
}
```

```
# Provide git sha during build on Windows
$env:GIT_SHA = (git rev-parse --short HEAD)
cargo build -p app --release
```

10.2 Docker Multi-Stage Builds

Even if you develop on Windows, production typically runs Linux containers. Use a multi-stage Dockerfile to:

- compile in a Rust builder image
- copy only the final binary into a minimal runtime image

Dockerfile (multi-stage, Rust + Debian slim)

Assume your binary crate is app and produces app (rename as needed).

```
# syntax=docker/dockerfile:1

FROM rust:1.85-bookworm AS builder
WORKDIR /src

# Cache dependencies first
COPY Cargo.toml Cargo.lock ./
COPY crates ./crates

# Build release
RUN cargo build -p app --release

# Runtime image
FROM debian:bookworm-slim AS runtime
WORKDIR /app

# Optional: add CA certs for outgoing HTTPS calls
RUN apt-get update && apt-get install -y --no-install-recommends ca-certificates \
    && rm -rf /var/lib/apt/lists/*
```

```
# Copy binary
COPY --from=builder /src/target/release/app /app/app

# Non-root user (recommended)
RUN useradd -m -u 10001 appuser
USER appuser

ENV RUST_LOG=info
EXPOSE 8080
ENTRYPOINT ["/app/app"]
```

Build and run (Windows PowerShell with Docker Desktop)

```
docker build -t rbe:latest .
docker run --rm -p 8080:8080 `
  -e RUST_LOG=info `
  -e APP_ENV=production `
  rbe:latest
```

Container-friendly runtime rules

- bind to `0.0.0.0` (not `127.0.0.1`)
- read config from environment variables
- log to stdout/stderr (no file logs inside containers)

10.3 Environment Configuration for Production

Principles

- **Config is data**: strongly typed, loaded once at startup

- **Secrets via environment or secret store**: never in code, never in git
- **Fail fast**: missing required settings should stop startup

Example environment variables

```
APP_ENV=production
APP_SERVER__HOST=0.0.0.0
APP_SERVER__PORT=8080
RUST_LOG=info

APP_DB__URL=postgres://user:pass@db:5432/app
APP_JWT_ACCESS_SECRET=very-long-secret
APP_JWT_REFRESH_SECRET=very-long-secret
```

Windows production-like run (PowerShell)

```powershell
$env:APP_ENV="production"
$env:APP_SERVER__HOST="0.0.0.0"
$env:APP_SERVER__PORT="8080"
$env:RUST_LOG="info"
$env:APP_DB__URL="postgres://user:pass@localhost:5432/app"

.\target\release\app.exe
```

Fail-fast config loader snippet

```rust
#[derive(Clone)]
pub struct Settings {
    pub host: String,
    pub port: u16,
    pub db_url: String,
}
```

```rust
fn must_env(key: &str) -> String {
    std::env::var(key).unwrap_or_else(|_| panic!("missing required env var: {key}"))
}

pub fn load_settings() -> Settings {
    Settings {
        host: must_env("APP_SERVER__HOST"),
        port: must_env("APP_SERVER__PORT").parse().expect("invalid port"),
        db_url: must_env("APP_DB__URL"),
    }
}
```

10.4 Graceful Shutdown and Lifecycle Management

A production server must:

- stop accepting new connections
- allow in-flight requests to finish (bounded)
- close DB pools / clients cleanly

Windows-first signal handling

On Windows, `Ctrl+C` is the standard signal for local and service control scenarios. In containers, SIGTERM is common (Linux). You can support both without changing your application API.

Graceful shutdown with Axum + Tokio

```rust
use std::net::SocketAddr;
use axum::Router;

async fn shutdown_signal() {
    // Windows and cross-platform: Ctrl+C
    let ctrl_c = async {
        tokio::signal::ctrl_c()
            .await
            .expect("failed to install Ctrl+C handler");
    };

    // On Unix you may also want SIGTERM; keep code portable by gating if needed.
    #[cfg(unix)]
    let terminate = async {
        use tokio::signal::unix::{signal, SignalKind};
        let mut sigterm = signal(SignalKind::terminate()).expect("sigterm handler");
        sigterm.recv().await;
    };

    #[cfg(not(unix))]
    let terminate = std::future::pending::<()>();

    tokio::select! {
        _ = ctrl_c => {},
        _ = terminate => {},
    }
}

pub async fn serve_with_shutdown(app: Router, addr: SocketAddr) {
    let listener = tokio::net::TcpListener::bind(addr).await.expect("bind failed");
    tracing::info!("listening on {}", listener.local_addr().unwrap());

    axum::serve(listener, app)
        .with_graceful_shutdown(shutdown_signal())
        .await
        .expect("server error");
}
```

Shutting down shared resources

For SQLx pools and many clients, dropping is enough; still, it is good to make shutdown explicit and bounded.

```
use sqlx::{Pool, Postgres};

pub async fn shutdown_db_pool(pool: Pool<Postgres>) {
    // Close waits for connections to return; bound it if you want strict shutdown time.
    pool.close().await;
}
```

Typical lifecycle in main

```
use std::{net::SocketAddr, sync::Arc};

#[tokio::main]
async fn main() {
    infrastructure::logging::init_logging("info");

    let cfg = infrastructure::config::load().expect("config load failed");
    let pool =
    ↪   infrastructure::db::create_pool(&cfg.db.url).await.expect("db connect failed");

    let state = Arc::new(api::state::AppState { db: pool.clone() });
    let app = api::routes::router(state);

    let addr: SocketAddr = format!("{}:{}", cfg.server.host,
    ↪   cfg.server.port).parse().unwrap();
    serve_with_shutdown(app, addr).await;

    shutdown_db_pool(pool).await;
    tracing::info!("shutdown complete");
}
```

10.5 Reverse Proxy Setup (Nginx / Caddy)

Reverse proxies are commonly responsible for:

- HTTPS termination
- gzip/brotli compression (optional)
- request size limits
- header normalization
- forwarding client information

10.5.1 Nginx (TLS termination + proxy to Axum)

Assume Axum listens on `127.0.0.1:8080` (internal) and Nginx exposes `443`.

```
server {
    listen 443 ssl http2;
    server_name api.example.com;

    ssl_certificate     /etc/ssl/certs/fullchain.pem;
    ssl_certificate_key /etc/ssl/private/privkey.pem;

    # Request size limit (tune per API)
    client_max_body_size 2m;
    location / {
        proxy_pass http://127.0.0.1:8080;

        proxy_http_version 1.1;
        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;

        # Timeouts (tune)
        proxy_connect_timeout 5s;
        proxy_send_timeout    30s;
        proxy_read_timeout    30s;
    }
}
```

10.5.2 Nginx: redirect HTTP to HTTPS

```
server {
    listen 80;
    server_name api.example.com;
    return 301 https://$host$request_uri;
}
```

10.5.3 Caddy (simpler HTTPS config)

Caddy can manage certificates automatically in many deployments. This example proxies to `127.0.0.1:8080`:

```
api.example.com {
    reverse_proxy 127.0.0.1:8080
    header {
        # Baseline security headers (adjust per needs)
        X-Content-Type-Options "nosniff"
        Referrer-Policy "no-referrer"
    }

    # Request size limit (Caddy v2 request body handling varies by handler; enforce at app
    ↪  too)
}
```

10.5.4 Reverse proxy and trusted forwarding

Security rule:

- trust `X-Forwarded-*` headers only from your proxy network
- do not accept `X-Forwarded-Proto=https` from arbitrary internet clients

10.6 Windows Commands (PowerShell)

```
# Release build
cargo build -p app --release

# Run with production-like env
$env:APP_ENV="production"
$env:APP_SERVER__HOST="0.0.0.0"
$env:APP_SERVER__PORT="8080"
$env:RUST_LOG="info"
.\target\release\app.exe

# Docker build/run (Windows + Docker Desktop)
docker build -t rbe:latest .
docker run --rm -p 8080:8080 -e RUST_LOG=info -e APP_ENV=production rbe:latest
```

Chapter outcomes

- You can produce optimized release binaries and enforce quality gates on Windows.
- You can containerize with multi-stage Docker builds while keeping runtime images small.
- You can run with environment-based configuration and fail-fast startup rules.
- You can shut down gracefully, preserving correctness under termination signals.
- You can deploy behind Nginx or Caddy with correct forwarding and security boundaries.

Conclusion

Rust Backend Engineering in Practice

In this booklet, we approached backend engineering from a production-first perspective. Not as a framework tutorial, but as a systems engineering discipline grounded in correctness, safety, performance, and operational clarity.
Rust does not merely offer a different syntax for backend development. It offers a different set of guarantees:

- Memory safety without a garbage collector.
- Concurrency without data races.
- Type-driven architecture that prevents entire classes of runtime errors.
- Explicit error handling instead of hidden failure paths.

These guarantees directly translate into backend systems that are:

- Predictable under load,
- Safer under concurrency,
- More transparent during debugging,
- More resilient in production.

From Framework Usage to Engineering Discipline

Throughout the chapters, we built a layered and modular backend architecture:

- Clear separation between `domain`, `api`, and `infrastructure`.
- Strongly typed routing and middleware pipelines.
- Explicit authentication and role-based authorization.
- Safe database integration with SQLx and compile-time query checks.
- Structured logging, request-scoped tracing, and metrics.
- Production-aware deployment, graceful shutdown, and reverse proxy configuration.

The objective was not only to show *how* to use Axum or SQLx, but to demonstrate *how to design backend systems that remain maintainable at scale*.
Rust encourages explicit thinking:

- Ownership clarifies lifecycle.
- Types clarify contracts.
- Traits clarify boundaries.
- Result types clarify failure.

This mindset is the foundation of production-grade backend systems.

Security as a First-Class Concern

Modern backends cannot treat security as an afterthought. In this booklet we addressed:

- Argon2 password hashing with salt and server-side pepper.
- JWT access and refresh token strategies with explicit claim design.
- Role-based authorization with enforceable guards.
- Request size limits and input validation.
- Secure headers and reverse proxy trust boundaries.
- Environment-based secret management.

Security in Rust is not automatic, but the language reduces the risk surface:

- No accidental null dereferences.
- No data races in safe code.
- No silent unchecked exceptions.

When combined with disciplined architecture, Rust becomes a powerful tool for secure backend development.

Observability and Operational Readiness

A backend that cannot be observed cannot be trusted in production.
We established:

- Structured logging using `tracing`.

- Request-scoped spans for correlation.
- Health and readiness endpoints aligned with operational models.
- A minimal but extensible metrics strategy.

Production readiness is not about feature count. It is about:

- Predictable startup,
- Controlled shutdown,
- Transparent runtime behavior,
- Measurable performance characteristics.

Rust integrates naturally with this mindset.

Testing as Design Validation

Testing was presented not as a secondary activity, but as a structural design requirement:

- Pure domain unit tests for fast validation.
- Integration tests using router-level invocation.
- Mock repositories and in-memory implementations.
- Authentication and RBAC verification.

Well-designed Rust code is inherently testable because:

- Dependencies are explicit.
- Traits define boundaries.
- Shared state is controlled and typed.

Testing becomes a natural extension of the architecture.

Deployment and Lifecycle Awareness

Production engineering does not end with compilation.
We addressed:

- Optimized release builds.
- Docker multi-stage images.
- Environment-driven configuration.
- Graceful shutdown handling.
- Reverse proxy integration with Nginx and Caddy.

These are not optional details. They are fundamental aspects of backend reliability.

What This Booklet Represents

This booklet is not an exhaustive encyclopedia of Rust web frameworks. It is a focused engineering guide that demonstrates:

- How to design a backend system using Rusts strengths.
- How to structure codebases for long-term maintainability.
- How to combine performance, safety, and clarity.
- How to prepare systems for real production environments.

Rust Backend Engineering is not about replacing one framework with another. It is about adopting a systems-oriented mindset in web development.

The Long-Term Perspective

As systems scale:

- Latency becomes visible.
- Concurrency issues become expensive.
- Memory inefficiencies accumulate.
- Operational mistakes multiply.

Rust does not eliminate engineering responsibility. But it enforces discipline at the language level, which dramatically reduces accidental complexity.
For backend systems that must remain stable, efficient, and secure over yearsnot monthsRust provides a foundation aligned with modern production requirements.

Final Thought

Backend engineering is ultimately about trust:

- Trust in correctness.
- Trust in performance.
- Trust in observability.
- Trust in security.

Rust, when used with architectural discipline, allows us to build systems worthy of that trust.
This concludes *Rust Backend Engineering*.

Appendices

Appendix A: Recommended Project Structure (Production Template)

Workspace Layout

```
rust-backend-engineering/

 Cargo.toml                (workspace)
 rust-toolchain.toml
 Dockerfile

 crates/
    app/                  (binary crate - composition root)
    api/                  (routing + handlers)
    domain/               (pure business logic)
    infrastructure/       (db, logging, config)
    shared/               (common types)

 migrations/
 .env (development only)
```

Workspace Cargo.toml

```toml
[workspace]
members = ["crates/*"]

[profile.release]
lto = true
codegen-units = 1
panic = "abort"
strip = "symbols"
```

Appendix B: Production Configuration Matrix

Minimal Required Environment Variables

```
APP_ENV=production
APP_SERVER__HOST=0.0.0.0
APP_SERVER__PORT=8080

APP_DB__URL=postgres://user:pass@db:5432/app

APP_JWT_ACCESS_SECRET=very-long-secret
APP_JWT_REFRESH_SECRET=very-long-secret
APP_PASSWORD_PEPPER=server-side-pepper
```

Windows PowerShell Setup

```powershell
$env:APP_ENV="production"
$env:APP_SERVER__HOST="0.0.0.0"
$env:APP_SERVER__PORT="8080"
$env:APP_DB__URL="postgres://user:pass@localhost:5432/app"
$env:APP_JWT_ACCESS_SECRET="very-long-secret"
```

```
$env:APP_JWT_REFRESH_SECRET="very-long-secret"
$env:APP_PASSWORD_PEPPER="strong-pepper"

.\target\release\app.exe
```

Appendix C: Secure Defaults Checklist

Authentication

- Argon2 for password hashing
- Access tokens short-lived
- Refresh token rotation implemented
- Secrets loaded from environment

HTTP Layer

- HTTPS enforced at proxy
- HSTS enabled after full HTTPS rollout
- Request body size limited
- CORS restricted to known origins

Observability

- Structured logging enabled
- Request ID propagation

- Health and readiness endpoints
- Metrics endpoint or structured metrics logs

Appendix D: Performance Tuning Guidelines

Connection Pool Tuning

- Max connections aligned with DB limits
- Avoid blocking inside async handlers
- Use timeouts for external dependencies

Release Optimization

```toml
[profile.release]
lto = true
codegen-units = 1
panic = "abort"
```

Common Anti-Patterns

- Blocking filesystem calls inside async handlers
- Long-lived locks across await points
- Storing unbounded in-memory state

Appendix E: Error Handling Patterns

Unified Application Error

```rust
pub enum AppError {
    BadRequest(&'static str),
    Unauthorized(&'static str),
    Forbidden(&'static str),
    NotFound,
    Conflict(&'static str),
    Internal(&'static str),
}
```

Mapping External Errors

```rust
impl From<sqlx::Error> for AppError {
    fn from(_: sqlx::Error) -> Self {
        AppError::Internal("database error")
    }
}
```

Appendix F: Testing Matrix

Recommended Coverage Strategy

- Domain logic: high coverage and deterministic tests
- Authentication logic: full flow validation
- Repository: integration tests with real DB
- Middleware: route-level tests

Windows Test Commands

```
cargo test
cargo test -p domain
cargo test -- --nocapture
```

Appendix G: Minimal Production Main Template

```rust
#[tokio::main]
async fn main() {
    infrastructure::logging::init_logging("info");

    let settings = infrastructure::config::load().expect("config load failed");
    let pool = infrastructure::db::create_pool(&settings.db_url)
        .await
        .expect("db connect failed");

    let state = std::sync::Arc::new(api::state::AppState { db: pool.clone() });

    let app = api::routes::router(state);

    let addr = format!("{}:{}", settings.host, settings.port)
        .parse()
        .unwrap();

    serve_with_shutdown(app, addr).await;

    pool.close().await;
}
```

Appendix H: Final Engineering Principles

- Favor explicit types over implicit assumptions.
- Fail fast at startup.
- Treat security as architecture, not configuration.
- Observe everything that matters.
- Design for graceful failure, not perfect uptime.
- Keep domain logic independent of infrastructure.

Closing Note

A production backend is defined not by framework choice, but by reliability guarantees. Rust enables those guarantees at the language level. Architecture and discipline make them sustainable.

This concludes the Appendices of *Rust Backend Engineering*.

References

Core Rust Language and Standard Library

- The Rust Programming Language (Official Rust Book), latest stable edition.
- Rust Reference (Language specification and semantics).
- Rust Standard Library Documentation (std).
- Rust Edition Guide (2021 and later editions).

Async Runtime and Concurrency

- Tokio Documentation (Asynchronous runtime, task scheduling, signals, timers).
- Futures and async/await model (Rust async ecosystem design).
- Tower Service Abstraction (Middleware and layered services).

Web Framework and HTTP Stack

- Axum Framework Documentation (Typed routing, extractors, middleware).

- Hyper HTTP Library Documentation (HTTP implementation and server model).
- Tower-HTTP Middleware Documentation (Tracing, CORS, compression, request ID).

Database Integration

- SQLx Documentation (Async, compile-time checked queries).
- SQLx CLI and Migration System Documentation.
- PostgreSQL Official Documentation (Transactions, indexing, constraints).

Authentication and Security

- Argon2 Password Hashing Specification (PHC winner design principles).
- RustCrypto Argon2 Crate Documentation.
- JSON Web Token (JWT) Specification (RFC 7519).
- JSON Web Algorithms (JWA) Specification (RFC 7518).
- JSON Web Signature (JWS) Specification (RFC 7515).
- OWASP Authentication Cheat Sheet.
- OWASP Password Storage Cheat Sheet.

HTTP Security and Reverse Proxy

- HTTP/1.1 Semantics and Content (RFC 9110 series).

- HTTP Strict Transport Security (HSTS) Specification (RFC 6797).
- Content Security Policy (CSP) Level 3 Specification.
- Nginx Official Documentation.
- Caddy Server Documentation.

Observability and Logging

- Tracing Crate Documentation (Structured logging and spans).
- Tracing-Subscriber Documentation (Filtering and formatting).
- Prometheus Exposition Format Documentation.
- OpenTelemetry Concepts (Traces, Metrics, Logs).

Testing and Quality Assurance

- Rust Testing Guide (Unit and integration testing).
- Tokio Testing Utilities.
- Tower Service Testing via ServiceExt.

Deployment and Containerization

- Docker Official Documentation (Multi-stage builds, image optimization).
- Debian Slim Base Image Documentation.
- Twelve-Factor App Methodology (Configuration and environment practices).

Operational and Production Engineering

- Site Reliability Engineering Principles (Service lifecycle, observability).
- Graceful Shutdown Patterns in Async Systems.
- Secure Coding Guidelines for Backend Services.

Final Note

All concepts, code patterns, and architectural practices in this booklet are derived from:

- Official Rust ecosystem documentation.
- Stable and widely adopted production libraries.
- Established security standards and RFC specifications.
- Modern backend engineering best practices.

This reference list represents authoritative and production-grade foundations for Rust backend engineering.