sessions/src/handlers.rs

```rust
async fn start_session(
  State(app): State<AppState>,
  Json(req): Json<StartSession>,
) -> Result<Json<Session>, ApiError> {
  let charger = app.repo
    .lock_charger(&req.charger_id).await?;
  Ok(Json(charger.start(req.driver)?))
}
```

```
$ cargo build --release
Compiling sessions
Finished release
$ cargo test ok ✓
```



Result<T, E> · Send + Sync · async/await · no GC

THE PRACTICAL GUIDE

Rust for Backend Developers

Safe, fast, production-ready services

Tokio · Axum · Tower · Serde · SQLx · Tonic · Redis
OpenTelemetry · Docker · Kubernetes

OWNERSHIP · ASYNC · APIS · DATA · PRODUCTION

THENEXBOOKS

Rust for Backend Developers

Safe, Fast, Production-Ready Services
With Tokio, Axum, SQLx, Tonic, and OpenTelemetry

First Edition

THENEXBOOKS

Rust for Backend Developers
Safe, Fast, Production-Ready Services

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Rust, its crate ecosystem, and the tools discussed in this book evolve quickly; always consult the official documentation for the versions you use.

Rust and Cargo are trademarks of the Rust Foundation. This book is not affiliated with, sponsored by, or endorsed by the Rust Foundation or the Rust Project. PostgreSQL, Redis, Docker, Kubernetes, OpenTelemetry, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement.

Voltrail, the running example in this book, is fictional, as are all operators, fleets, providers, and people mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

Backend services live a demanding life. They run for months without restarts, handle thousands of concurrent connections, parse untrusted input from the whole internet, talk to databases, caches, queues, and other services that fail in creative ways, and are expected to stay fast and correct while doing all of it. Every class of bug that slips through — a data race, a null dereference, a buffer overrun, a forgotten error, a leaked connection — eventually turns into an incident.

Rust was designed to rule out many of those bugs before the program ever runs. Its ownership model makes memory safety and data-race freedom compile-time guarantees rather than hopes. Its type system makes invalid states hard to represent and errors impossible to ignore silently. And it does this without a garbage collector, so services get predictable latency and a small, steady memory footprint. The price is a steeper learning curve, especially the first weeks with the borrow checker.

This book is about paying that price once and collecting the benefits for the life of your services. It is not a language reference. It is a practical guide to building production backend services in Rust: HTTP and gRPC APIs, database access, caching, messaging, observability, resilience, security, deployment, and architecture — with the language features introduced exactly when a real service needs them, and with honest discussion of where Rust makes things harder.

Who This Book Is For

- **Backend developers** coming from Go, Java, C#, Python, Node.js, or PHP who want to build services in Rust.
- **Go developers** looking for a companion to *Go (Golang) for Backend Developers*: this book follows the same six-part structure, so you can compare approaches chapter by chapter.
- **Rust developers** who know the language but want solid, production-oriented answers for web services, data access, and operations.
- **Tech leads and architects** evaluating Rust for performance- and reliability-critical services.

What You Will Learn

- Cargo, workspaces, ownership, borrowing, lifetimes, traits, generics, and idiomatic error handling — explained for service developers.
- Testing with unit, integration, doc, and property tests, and benchmarking.
- Threads, `Send` and `Sync`, async Rust, the Tokio runtime, channels, shared state, cancellation, and backpressure.
- HTTP APIs with Axum and Tower middleware, JSON with Serde, validation, authentication, and authorization.
- gRPC with Tonic, configuration, and structured logging with `tracing`.

- PostgreSQL with SQLx, transactions and migrations, Redis caching, events, background jobs, and the outbox pattern.
- Observability with OpenTelemetry, profiling and performance, resilience patterns, graceful shutdown, and security.
- Containers, Kubernetes, CI/CD, large-codebase architecture, and two end-to-end case studies.

How This Book Is Organized

Part I — Rust Foundations for Backend Engineers (Chapters 1–6) covers the language and tooling you need for services. **Part II — Concurrency and Async** (Chapters 7–10) explains threads, async Rust, Tokio, and the concurrency patterns services rely on. **Part III — Building HTTP and RPC APIs** (Chapters 11–16) builds REST and gRPC APIs with Axum and Tonic. **Part IV — Data and Messaging** (Chapters 17–20) connects services to PostgreSQL, Redis, and a message broker. **Part V — Production Readiness** (Chapters 21–25) makes services observable, fast, resilient, secure, and deployable. **Part VI — Architecture and Case Studies** (Chapters 26–28) structures a large codebase and builds two complete services. The appendices provide a cheat sheet, a service template and crate reference, a production checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Voltrail

Throughout the book we build **Voltrail**, a fictional network of electric-vehicle charging stations. Drivers find chargers and start charging sessions from a mobile app, station operators manage their sites and tariffs, fleet customers get consolidated invoices, and thousands of chargers stream meter readings back to the platform every few seconds. That gives us everything a real backend faces: a public REST API, a state machine with money attached, high-volume telemetry ingestion, payments through an external provider, events and background jobs, and strict requirements for correctness and uptime. The code is a Cargo workspace named `voltrail` with services such as `api-gateway`, `sessions`, `telemetry`, `billing`, and `notifications`.

Versions and Conventions

Examples target stable Rust with the 2024 edition and recent versions of the main crates: Tokio, Axum, Tower, Serde, SQLx, Tonic, and the `tracing` ecosystem. The crate ecosystem moves quickly, so where an API changed recently we say so, and we encourage you to check each crate's documentation for the exact versions you pin. Nightly-only features are labeled as such.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake or a behavior that can cause bugs, data loss, or outages in production.

Code appears in `monospace`. Long lines are wrapped to fit the page, money is stored as integer minor units, and production code paths return `Result` rather than calling `unwrap()`.

Contents

Preface

PART I — RUST FOUNDATIONS FOR BACKEND ENGINEERS

1 Why Rust for Backend Development

2 Tooling: Cargo, Crates, Workspaces, and Project Layout

3 Ownership, Borrowing, and Lifetimes for Service Developers

4 Types, Structs, Enums, Traits, and Generics

5 Error Handling Done Right: Result, the ? Operator, thiserror, and anyhow

6 Testing Fundamentals: Unit, Integration, Doc Tests, Property Tests, and Benchmarks

PART II — CONCURRENCY AND ASYNC

7 Threads, Send and Sync, and Fearless Concurrency

8 Async Rust and the Tokio Runtime

9 Channels, Shared State, and Synchronization in Async Code

10 Concurrency Patterns: Task Sets, Pipelines, Cancellation, and Backpressure

PART III — BUILDING HTTP AND RPC APIS

11 HTTP Services with Axum: Routers, Handlers, Extractors, and State

12 Middleware with Tower and tower-http: Layers, Services, and the Request Lifecycle

13 JSON with Serde, Validation, and REST API Design

14 Authentication and Authorization

15 gRPC with Tonic and Protocol Buffers

16 Configuration and Structured Logging with tracing

PART IV — DATA AND MESSAGING

17 Working with PostgreSQL Using SQLx

18 Transactions, Migrations, and Data-Access Patterns

19 Caching with Redis

20 Messaging, Events, and Background Jobs

PART V — PRODUCTION READINESS

21 Observability: Metrics, Tracing, and OpenTelemetry

22 Performance: Profiling, Allocations, and Zero-Cost Abstractions in Practice

23 Resilience: Timeouts, Retries, Rate Limiting, Load Shedding, and Graceful Shutdown

24 Security for Rust Services

25 Containers, Kubernetes, and CI/CD for Rust

PART VI — ARCHITECTURE AND CASE STUDIES

26 Structuring Large Rust Codebases: Workspaces, Modules, and Hexagonal Design

27 Case Study: Building the Voltrail Charging Session Service End to End

28 Case Study: Adding an AI Assistant Backend in Rust

APPENDICES

A Rust Backend Cheat Sheet

B Service Template and Crate Reference

C Production Checklist

D Glossary

PART I

Rust Foundations for Backend Engineers

Why Rust fits backend services, the Cargo toolchain and project layout, ownership and borrowing explained for service code, the type system, idiomatic error handling, and testing — everything a service needs from the language.

1 · Why Rust for Backend Development
2 · Cargo, Crates, and Workspaces
3 · Ownership, Borrowing, and Lifetimes
4 · Types, Traits, and Generics
5 · Error Handling Done Right
6 · Testing Fundamentals

CHAPTER 1 · PART I — RUST FOUNDATIONS FOR BACKEND ENGINEERS

Why Rust for Backend Development

Rust promises memory safety without a garbage collector, predictable performance, and a type system that turns whole classes of production incidents into compile errors. This chapter explains what those promises actually mean for a backend team, what they cost, and how the rest of the book builds a real service — the Voltrail EV charging platform — on top of them.

The Problem Backend Services Actually Have

Most backend developers do not wake up in the morning wishing their language were faster. They wake up to a page: a service is leaking memory and gets killed by the orchestrator every six hours; a `NullPointerException` fires on a code path nobody tested; two goroutines or threads race on a map and corrupt it under load; the p99 latency graph shows periodic spikes that line up suspiciously with garbage-collection cycles; or a refactor six months ago quietly broke the handling of an enum variant that only shows up on the last day of the month.

These problems share a theme. They are not about raw throughput; they are about **correctness under concurrency and predictability under load**. A backend service is a long-running process that handles untrusted input from many clients at once, holds connections to databases and brokers, and must keep doing that for weeks without restarts. The failures that hurt are the ones that appear only in production, only under load, and only intermittently.

Rust was designed by people who build exactly this kind of long-running, concurrent, safety-critical software (a web browser engine was the original motivating project). Its core ideas — ownership, borrowing, algebraic data types, and traits — map surprisingly well onto the daily problems of backend engineering. This book is about using those ideas deliberately, not about writing clever code.

What Rust Gives You

Memory safety without a garbage collector

In languages such as Java, C#, Go, Python, and JavaScript, memory safety comes from a garbage collector (GC) or reference counting managed by a runtime. You allocate freely, and the runtime figures out later when memory is no longer reachable. This is convenient and, for most services, perfectly fine. It does have costs: the collector needs CPU time, it needs headroom (heaps are usually sized well above the live data set), and depending on the collector design, it can introduce pauses or latency variance that you have to tune around.

In C and C++, there is no collector; you manage memory manually, and the compiler trusts you. That gives excellent control and has produced decades of use-after-free bugs, double frees, buffer overflows, and data races — the classic sources of security vulnerabilities.

Rust takes a third path. Every value has exactly one **owner**. When the owner goes out of scope, the value is dropped and its memory (and any other resource it holds, such as a socket or a database connection) is released immediately and deterministically. You can lend access to a value through **references**, and the compiler's *borrow checker* proves, at compile time, that no reference outlives the value it points to and that nobody mutates data while someone else is reading it. There is no runtime collector, and yet safe Rust code cannot produce a dangling pointer or a data race. We spend all of Chapter 3 on how this works and how to live with it comfortably.

The practical effect for a service is that memory usage tracks the live working set closely, and releases happen at predictable points in the code. That does not automatically mean "less memory" or "lower latency" for your particular service — allocation patterns, data structures, and I/O dominate most real workloads — but it removes one large source of variance. Always measure your own service before and after; we show how in Chapter 22.

A type system that models your domain

Rust's type system is where many backend developers find the most day-to-day value, often more than in the performance story. Two features stand out.

First, **enums carry data**. A Rust `enum` is a *sum type*: a value is exactly one of several variants, and each variant can hold its own fields. Combined with `match`, which the compiler checks for exhaustiveness, this lets you model state machines so that invalid states are unrepresentable and every state is handled.

Second, **there is no null**. A value that may be absent has type `Option<T>`, and an operation that may fail returns `Result<T, E>`. You cannot forget to check; the compiler will not let you use the inner value until you have handled the other case. Here is a small taste from the Voltrail domain:

crates/voltrail-core/src/station.rs

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum ChargerStatus {
    Available,
    Charging,
    Faulted,
    Offline,
}

/// Can a driver start a new session on this charger right now?
pub fn can_start_session(status: ChargerStatus) -> bool {
    match status {
        ChargerStatus::Available => true,
        ChargerStatus::Charging
        | ChargerStatus::Faulted
```

```
        | ChargerStatus::Offline => false,
    }
}
```

If a colleague later adds a `Reserved` variant to `ChargerStatus`, this function stops compiling until someone decides what "reserved" means for session starts. In many languages that decision would be made silently by a `default:` branch — or not made at all. The compiler error looks roughly like this:

```
error[E0004]: non-exhaustive patterns: `ChargerStatus::Reserved` not covered
  --> crates/voltrail-core/src/station.rs:12:11
   |
12 |     match status {
   |           ^^^^^^ pattern `ChargerStatus::Reserved` not covered
```

That is the kind of error you want: it appears on your laptop, it names the exact case, and it costs you thirty seconds instead of an incident review.

Fearless concurrency

Backend services are concurrent by nature. Rust extends ownership into concurrency through two marker traits, `Send` (a value may be moved to another thread) and `Sync` (a value may be shared by reference between threads). The compiler infers them automatically for your types and refuses to compile code that would share non-thread-safe data across threads. A plain `HashMap` cannot be mutated from two tasks at once; you must wrap it in a `Mutex`, an `RwLock`, or use a channel. The phrase the Rust community uses is "fearless concurrency": not that concurrency becomes easy, but that data races become compile errors instead of heisenbugs. Chapter 7 covers this in depth, and Part II builds on it with async Rust and Tokio.

> **NOTE**
>
> Rust prevents *data races* (unsynchronized concurrent access where at least one access is a write). It does not prevent every *race condition* — for example, two requests that both read "charger is available" and both try to start a session. Those are logic bugs that you still solve with transactions, locks, or idempotency keys, as we show in Chapters 18 and 20.

Zero-cost abstractions and predictable performance

"Zero-cost abstraction" is a principle Rust inherits from C++: you should not pay at runtime for abstractions you do not use, and the abstractions you do use should compile down to code roughly as efficient as what you would write by hand. Iterators are the classic example. A chain of `filter`, `map`, and `sum` is compiled, through monomorphization and inlining, into a tight loop without intermediate collections or virtual calls.

crates/billing/src/energy.rs

```
use voltrail_core::MeterReading;

/// Total energy delivered by readings above a noise threshold.
pub fn billable_energy_wh(readings: &[MeterReading], min_power_w: u32) -> u64 {
    readings
        .iter()
        .filter(|r| r.power_w >= min_power_w)
        .map(|r| r.energy_wh)
        .sum()
}
```

This reads like high-level code, yet nothing here allocates, and there is no dynamic dispatch. Generics work the same way: a generic function is specialized for each concrete type it is called with, so there is no boxing and no runtime type check. When you *do* want dynamic dispatch — for example, to swap a payment provider at runtime — you opt into it explicitly with `dyn Trait`, and you can see the cost in the type. Chapter 4 explains when to choose each.

The honest version of the performance story is this: Rust makes it *possible* to write services with low, stable latency and a small memory footprint, because the language adds very little overhead of its own. Whether your service is faster than its Go or Java equivalent depends on what it does. A service that spends most of its time waiting on PostgreSQL will be bound by PostgreSQL in any language. Measure, do not assume.

A serious backend ecosystem

A few years ago "Rust for web services" meant accepting immature libraries. That has changed. The crates we use in this book are widely deployed and actively maintained:

Need	Crates used in this book	Chapter
Async runtime	`tokio`, `futures`, `tokio-util`	8–10
HTTP server	`axum`, `tower`, `tower-http`, `hyper`	11–12
Serialization	`serde`, `serde_json`	13
gRPC	`tonic`, `prost`	15
Logging and tracing	`tracing`, `tracing-subscriber`	16, 21
PostgreSQL	`sqlx`	17–18
Redis	`redis`, `deadpool-redis`	19
Errors	`thiserror`, `anyhow`	5
Testing	`proptest`, `criterion`, `cargo-nextest`	6

Several large technology companies run Rust in production for networking, storage, and infrastructure services, and major operating systems and browsers now include Rust code. You will not be the first team to put Rust behind a load balancer.

What Rust Costs You

A book that only lists benefits is a brochure. Rust has real costs, and a team adopting it should plan for them explicitly.

The learning curve is real

Ownership and borrowing are new concepts for developers coming from garbage-collected languages. For the first few weeks, you will write code that you are sure is correct and the compiler will reject it. Sometimes the compiler is catching a genuine bug; sometimes it is being conservative and you need to restructure the code so that the proof is easier. Both experiences are part of learning, and both become rare with practice. The biggest mistake new teams make is fighting the borrow checker with `.clone()` everywhere and `Rc<RefCell<T>>` soup instead of learning the handful of patterns that make ownership natural. Chapter 3 is dedicated to those patterns, framed around service code rather than toy examples.

Compile times

Rust compiles slowly compared to Go. The compiler does a lot of work — borrow checking, monomorphizing generics, aggressive optimization in release mode — and large dependency trees add up. Incremental debug builds of a single crate are usually fast enough for a tight edit-compile-test loop, but a clean release build of a service with a few hundred transitive dependencies takes noticeably longer. You manage this with workspace structure, `cargo check` during development, build caching in CI, and sensible dependency hygiene; Chapters 2 and 25 cover the practical techniques.

Async adds its own complexity

Async Rust is powerful and efficient, but it has sharp edges that do not exist in Go's goroutine model: futures must be polled by a runtime, blocking calls inside async code stall a worker thread, holding a standard-library mutex guard across an `.await` is a common trap, and some error messages involving `Send` bounds on futures are long. None of this is insurmountable, and Part II walks through it carefully. But budget time for it.

Smaller ecosystem in some niches

For HTTP, gRPC, PostgreSQL, Redis, and serialization, the ecosystem is mature. For some enterprise integrations — a niche SOAP service, a vendor SDK that only ships for Java and .NET, a particular reporting library — you may find nothing, or an unofficial crate maintained by one person. Check your critical integrations before committing, and remember that calling a plain HTTP or gRPC API needs no vendor SDK.

> **WARNING**
>
> Do not rewrite a working service in Rust just because Rust is interesting. Rewrites are expensive, they reintroduce bugs the old system had already fixed, and they delay features. Start Rust where it pays off clearly: a new service, a hot path with latency problems, a component where memory usage drives infrastructure cost, or a piece of code where correctness failures are expensive.

When to Choose Rust — and When Not To

Language choice is a trade-off, not a verdict. The table below summarizes how we think about it for backend work. It deliberately avoids benchmark numbers; the right numbers are the ones you measure on your own workload.

Situation	Rust is a strong fit	Consider alternatives
Latency-sensitive path	Stable tail latency matters	Latency is dominated by I/O
High-volume ingestion	Many messages, tight memory	Low, bursty volume
Complex domain rules	State machines, money, protocols	Simple CRUD over a database
Team experience	Time to learn is budgeted	Deadline in two weeks
Integrations	HTTP, gRPC, SQL, Redis, Kafka	Vendor SDK only in Java/.NET
Deployment footprint	Small containers, edge, sidecars	Footprint is not a constraint

If you come from Go, Rust will feel stricter and more expressive: Go optimizes for simplicity and fast compilation, Rust for correctness and control. Many organizations use both — Go for the bulk of CRUD services, Rust for the components where its guarantees pay for themselves. If you come from Java or C#, you will miss some runtime reflection and a large framework like Spring or ASP.NET Core, but you will find that traits, generics, and macros cover most of what those frameworks do, with the wiring visible in the code rather than hidden in annotations and containers. If you come from Python, Node.js, or PHP, the biggest shift is that the compiler asks you to be explicit about types and ownership up front, in exchange for catching errors that would otherwise surface in production.

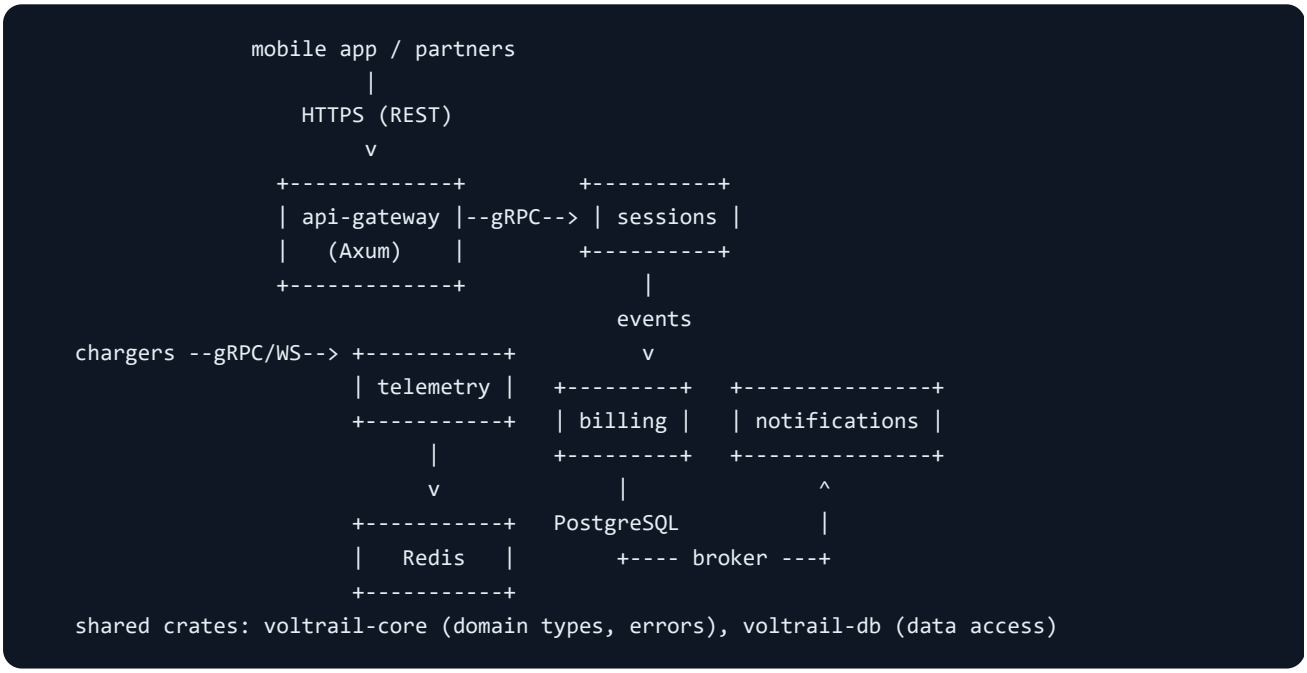
Meet Voltrail: The Running Example

Throughout this book we build pieces of **Voltrail**, a fictional network of electric-vehicle charging stations operating in several countries. We chose an EV charging network because it exercises nearly everything a modern backend has to handle:

- **A public REST API** used by a mobile app for drivers: find nearby stations, see which chargers are available, start and stop charging sessions, and view invoices.

- **A real state machine**: a charging session moves through `Pending`, `Active`, `Completed`, or `Failed`, and chargers report `Available`, `Charging`, `Faulted`, or `Offline`. Illegal transitions must be impossible.
- **High-volume telemetry**: every active charger streams meter readings — energy delivered and current power — that must be ingested, validated, and aggregated without falling behind.
- **Money**: tariffs, idle fees, invoices, and payments through a fictional payment provider called PayPlume. Money is always stored in integer minor units (cents) with an ISO currency code, never as floating point.
- **Multiple tenants**: station operators such as "Northgate Parking" own sites and set tariffs; fleet customers such as "Fjordline Fleet" have many drivers billed to one account.
- **Events and background work**: `session.started`, `session.completed`, and `charger.faulted` events drive notifications, billing, and support dashboards.

The system is organized as a Cargo workspace named `voltrail`. Chapter 2 builds its skeleton; later chapters fill it in.



The domain types are shared through `voltrail-core`. You will see the same names in every chapter: `Station`, `Charger`, `ConnectorType`, `ChargerStatus`, `Driver`, `Operator`, `Tariff`, `ChargingSession`, `SessionState`, `MeterReading`, `Invoice`, and `Payment`. Here is a first sketch of the session type, which Chapter 4 refines:

crates/voltrail-core/src/session.rs

```
use chrono::{DateTime, Utc};
use uuid::Uuid;

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum SessionState {
    Pending,
```

```
    Active,
    Completed,
    Failed,
}

#[derive(Debug, Clone)]
pub struct ChargingSession {
    pub id: Uuid,
    pub charger_id: Uuid,
    pub driver_id: Uuid,
    pub state: SessionState,
    pub started_at: Option<DateTime<Utc>>,
    pub ended_at: Option<DateTime<Utc>>,
    pub energy_wh: u64,
    pub cost_minor: i64,
}
```

Notice how much the types already say. `started_at` is an `Option` because a pending session has not started yet. Energy is an unsigned `u64` in watt-hours because it cannot be negative. Cost is a signed `i64` in minor units because refunds and adjustments can make amounts negative. None of this is enforced by comments; it is enforced by the compiler.

Where Rust pays off in Voltrail

It is worth being concrete about why a team would pick Rust for each Voltrail component, because the reasons differ:

- **`telemetry`** is the throughput-bound service. Thousands of chargers each send a reading every few seconds, and the service must parse, validate, and batch them without building up memory. Here the absence of GC pauses, compact data layout, and cheap async tasks matter most.
- **`sessions`** is the correctness-bound service. Starting a session twice, stopping a session that never started, or billing a failed session are expensive mistakes. Enums, exhaustive matching, and state types that cannot be misused carry most of the weight here.
- **`billing`** is where money lives. Integer minor units, checked arithmetic that refuses to overflow silently, and newtypes that stop you from adding euros to Norwegian kroner are the relevant features.
- **`api-gateway`** is a fairly ordinary REST service. Rust is not dramatically better than other languages for this kind of code, but sharing `voltrail-core` types with the other services means request validation and domain rules are written once.
- **`notifications`** is a background worker that mostly waits on external providers. Any language would do; we write it in Rust for consistency and to demonstrate worker patterns in Chapter 20.

This mix is typical. In a real organization you would probably not write every service in Rust on day one; you would start with the component where the argument is strongest — here, `telemetry` or `sessions` — and expand once the team is comfortable.

Common Misconceptions

Teams evaluating Rust often arrive with a few beliefs that are worth correcting early, because they shape expectations in unhelpful ways.

"Rust is a systems language, so it is too low-level for web services." Rust can go low-level, but idiomatic service code rarely does. A typical Axum handler reads much like a handler in a modern TypeScript or Kotlin framework: extract a JSON body, call a repository, return a response. The low-level control is available when you need it and invisible when you do not.

"Safe Rust means no bugs." Safe Rust rules out memory-safety bugs and data races. It does not rule out wrong business logic, deadlocks, unbounded queues, slow SQL queries, panics from an out-of-bounds index, or integer overflow in release builds (which wraps by default unless you use checked arithmetic or enable overflow checks). You still need tests, reviews, and observability.

"You need `unsafe` to get good performance." Application code almost never needs `unsafe`. It lives in carefully audited low-level crates — the async runtime, some data structures, FFI bindings — which expose safe APIs. Chapter 24 discusses how to audit dependencies that use it.

"Everything has to be cloned to satisfy the borrow checker." Beginners clone a lot, and that is fine while learning; cloning a small struct or an `Arc` is cheap. But idiomatic Rust passes references for read-only access, moves values when ownership changes hands, and shares long-lived state through `Arc`. Once these habits settle in, clones become a deliberate choice rather than a reflex.

Setting Up Your Toolchain

Rust is installed and updated with `rustup`, the official toolchain manager. It installs the compiler (`rustc`), the build tool and package manager (`cargo`), the standard library, and optional components such as the formatter and linter. Follow the instructions on the official Rust website for your platform; on Linux and macOS it is a single shell command, and on Windows there is an installer (you will also need the Microsoft C++ build tools for linking).

```
$ rustup --version
$ rustup update stable
$ rustup component add rustfmt clippy
$ rustc --version
$ cargo --version
```

This book targets **stable Rust with the 2024 edition**, which was stabilized with Rust 1.85 in early 2025. Editions are opt-in language revisions that let Rust evolve without breaking old code: each crate declares its edition in `Cargo.toml`, and crates of different editions link together seamlessly. Any recent stable toolchain will compile the code in this book. We never rely on nightly-only features; when we mention one, we label it as unstable.

> **TIP**
>
> Install `rust-analyzer` in your editor (VS Code, the JetBrains IDEs via their Rust plugin, Neovim, Helix, and Zed all support it). Inline type hints, go-to-definition into dependencies, and instant borrow-checker feedback shorten the learning curve more than any other single tool.

Your First Voltrail Endpoint

Let us finish the chapter by running a tiny HTTP service, so you have seen the whole loop — from `cargo new` to a JSON response — before we dig into the details. Do not worry about understanding every line; each piece is explained later in the book.

```
$ cargo new voltrail-hello
$ cd voltrail-hello
$ cargo add tokio --features full
$ cargo add axum
$ cargo add serde --features derive
$ cargo add serde_json
```

voltrail-hello/src/main.rs

```
use axum::{routing::get, Json, Router};
use serde::Serialize;

#[derive(Serialize)]
struct Health {
    status: &'static str,
    service: &'static str,
}

async fn health() -> Json<Health> {
    Json(Health {
        status: "ok",
        service: "voltrail-hello",
    })
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let app = Router::new().route("/api/v1/health", get(health));

    let listener = tokio::net::TcpListener::bind("127.0.0.1:8080").await?;
    println!("listening on {}", listener.local_addr()?);
    axum::serve(listener, app).await?;
    Ok(())
}
```

```
$ cargo run
listening on 127.0.0.1:8080
```

```
$ curl -s http://127.0.0.1:8080/api/v1/health
{"status":"ok","service":"voltrail-hello"}
```

A few things are worth noticing even at this stage. `#[tokio::main]` is a macro that starts the Tokio async runtime and runs `main` inside it. The handler `health` is an ordinary `async fn`; Axum figures out from its return type how to turn the value into an HTTP response. `#[derive(Serialize)]` generates JSON serialization code at compile time — there is no reflection at runtime. And `main` returns a `Result`, so the `?` operator propagates a failure to bind the port instead of panicking. Using `Box<dyn std::error::Error>` as the error type is fine for a demo; Chapter 5 replaces it with proper error types.

Run `cargo build --release` and look in `target/release`: you get a single native executable with no runtime to install alongside it. That property makes Rust services pleasant to containerize, which we exploit in Chapter 25 with minimal container images.

How This Book Is Organized

The book is divided into six parts that follow the life of a production service:

1. **Part I — Rust Foundations (Chapters 1–6)**: tooling, ownership, types and traits, error handling, and testing — the language skills every later chapter assumes.
2. **Part II — Concurrency and Async (Chapters 7–10)**: threads, `Send` and `Sync`, Tokio, channels, shared state, cancellation, and backpressure.
3. **Part III — HTTP and RPC APIs (Chapters 11–16)**: Axum, Tower middleware, Serde and validation, authentication, gRPC with Tonic, configuration, and structured logging.
4. **Part IV — Data and Messaging (Chapters 17–20)**: PostgreSQL with SQLx, transactions and migrations, Redis caching, events, and the outbox pattern.
5. **Part V — Production Readiness (Chapters 21–25)**: observability, performance, resilience, security, containers, and CI/CD.
6. **Part VI — Architecture and Case Studies (Chapters 26–28)**: structuring large codebases, the Voltrail session service end to end, and an AI assistant backend.

Part I moves briskly. We assume you already know what an HTTP request, a SQL join, and a unit test are, and we spend our words on what is different in Rust. If you already write Rust, skim Part I for the Voltrail domain types and conventions, then jump to whichever part matches your current problem. Every code example is written against the same workspace, so later chapters build on names and patterns introduced earlier.

One convention before we start: code in this book avoids `unwrap()` on production paths. You will see it in tests and in clearly marked quick demos, but real handlers return errors instead of crashing the process. Treating a panic in a request handler as a bug, not as error handling, is one of the habits that separates a Rust prototype from a Rust service.