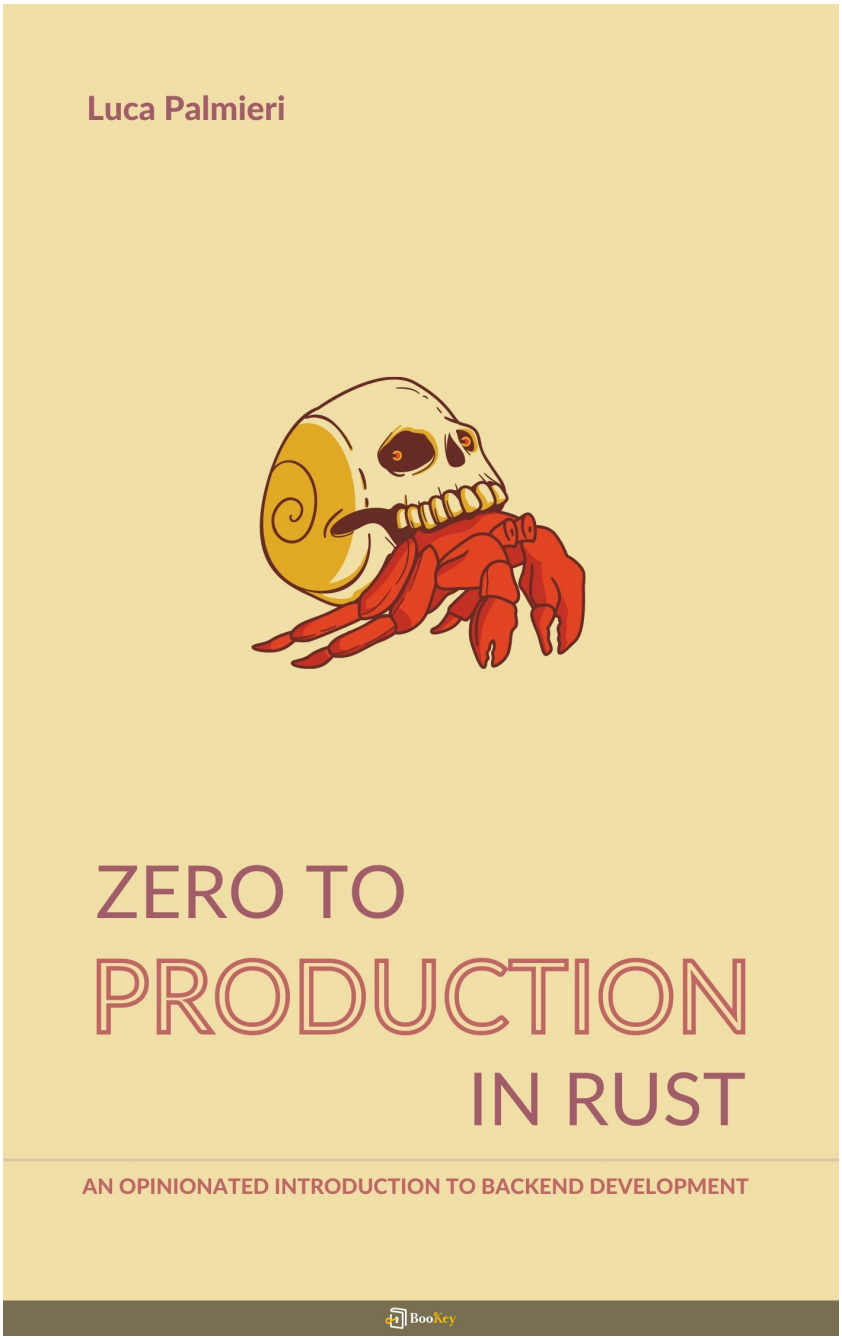
Zero To Production In Rust PDF (Limited Copy)

Luca Palmieri



Zero To Production In Rust Summary

Master Backend Development in Rust Through Hands-On API Building

Written by New York Central Park Page Turners Books Club

About the book

In "Zero To Production In Rust," author Luca Palmieri guides readers through the exciting landscape of backend development, focusing on building a robust email newsletter backend API using the Rust programming language. This practical guide is not only for those new to backend development but also for seasoned programmers looking to enhance their skills in Rust.

The journey begins with an introduction to the Rust ecosystem, notably its crates — packages that extend the language's capabilities. Palmieri emphasizes the importance of understanding these tools as they form the building blocks for the application. Readers will learn to design modular applications, promoting a clean architecture that allows for easier maintenance and expansion.

As the narrative progresses, the book delves into testing strategies. Palmieri discusses the significance of testing in ensuring the reliability of web applications, covering techniques for unit and integration testing. This section equips readers with the knowledge to implement rigorous testing phases in their development process.

An essential aspect of backend development is modeling the domain effectively. Utilizing Rust’s powerful type system, the book guides readers

through creating data models that accurately represent the application's requirements, providing a solid foundation for building a fault-tolerant system.

Furthermore, the guide emphasizes the need for effective application monitoring. Readers will explore how to collect logs, traces, and metrics, which are crucial for identifying issues and optimizing performance in real time. The implementation of monitoring practices is framed within the context of maintaining application health, ensuring that readers understand its critical importance.

Lastly, the book covers the establishment of a continuous integration and deployment (CI/CD) pipeline tailored specifically for Rust projects. This section empowers readers to automate testing and deployment processes, ensuring that their production systems are not only efficient but also robust and responsive to changes.

By the end of this enlightening journey, readers will have acquired a comprehensive understanding of backend development in Rust, leaving them well-prepared to build reliable and efficient systems in their future endeavors. The blend of practical guidance and foundational theory makes this resource invaluable for anyone looking to thrive in the world of backend programming.

About the author

In "Zero To Production In Rust," Luca Palmieri embarks on a journey to equip programmers, regardless of their experience level, with the skills necessary to build robust, scalable applications using the Rust programming language. The book is divided into chapters that each focus on essential aspects of software development, marrying theoretical knowledge with practical application.

Luca introduces readers to the fundamental concepts of Rust, emphasizing its unique memory management capabilities that prevent common bugs seen in other languages. His approachable teaching style breaks down complex topics into digestible segments, ensuring that even those new to programming can follow along with ease.

As the chapters progress, Luca employs real-world examples and projects to illustrate the power of Rust in producing production-ready applications. He covers critical areas such as systems design, asynchronous programming, and the Rust ecosystem, which includes libraries and frameworks instrumental in application development.

Through the narrative, Luca also highlights the importance of solid software engineering principles, encouraging readers to think critically about design patterns and the architecture of their applications. He incorporates best

practices that involve testing, debugging, and deployment, empowering developers to be confident in their coding and problem-solving skills.

In summary, "Zero To Production In Rust" serves as both a practical guide and an educational resource, presenting Rust not just as a programming language, but as a tool for professional growth and innovation in software development. By the conclusion, readers will not only have gained technical proficiency but also a deeper appreciation for the methodologies and philosophies that drive successful programming and systems design.

Try Bookey App to read 1000+ summary of world best books

Unlock 1000+ Titles, 80+ Topics

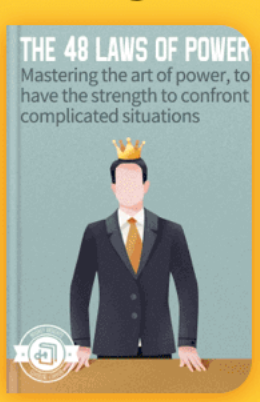
New titles added every week

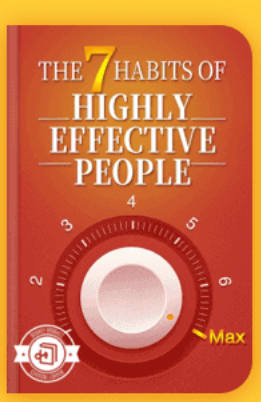
Brand | Leadership & Collaboration | Time Management | Relationship & Communication

ness Strategy | Creativity | Public | Money & Investing | Know Yourself | Positive P

Entrepreneurship | World History | Parent-Child Communication | Self-care | Mind & Spi

Insights of world best books





Free Trial with Bookey

Summary Content List

Chapter 1 Summary: 1 Getting Started

Chapter 1: Getting Started

In the realm of programming, effective tools are just as vital as syntax. Rust, a systems programming language, places a high emphasis on its tooling, which significantly influences its adoption among developers. This chapter introduces essential tools that streamline the Rust development journey, showcasing both officially supported and community-driven resources.

1.1 Installing the Rust Toolchain

The installation of Rust can be approached in several ways, but the preferred method is through **rustup**, which offers comprehensive toolchain management. Rustup enables developers to effortlessly switch between different versions of Rust and manage various compilation targets.

1.1.1 Compilation Targets

To run Rust code on different platforms like Linux or macOS, the Rust compiler uses different backends, resulting in a myriad of compilation targets categorized into three tiers. These tiers vary based on the guarantees they provide regarding dependencies.

1.1.2 Release Channels

Developers can choose from different release channels, including stable, beta, and nightly versions, depending on their need for stability or access to the latest features. The stable channel, updated every six weeks, is recommended for production applications, while the beta and nightly channels cater to those who wish to test upcoming enhancements.

1.1.3 What Toolchains Do We Need?

When installing Rustup, it automatically sets up the latest stable compiler for the operating system. If developers require specific features from the nightly version, they can easily install it using the command: `rustup toolchain install nightly --allow-downgrade`, ensuring compatibility with other components.

1.2 Project Setup

Once the toolchain is in place, developers will primarily interact with **Cargo** , Rust's package manager, essential for creating, managing, and maintaining projects. Using the command `cargo new zero2prod`, developers can quickly initiate a project structure, complete with an integrated Git repository.

1.3 IDEs

Choosing a suitable Integrated Development Environment (IDE) is crucial for productivity, especially for beginners. The primary options available include **rust-analyzer**, which works with various code editors, and **IntelliJ Rust**, part of JetBrains' suite, which was regarded as superior as of October 2021, despite rust-analyzer's flexibility.

1.4 Continuous Integration

To maintain team efficiency, incorporating Continuous Integration (CI) systems is critical. CI pipelines automate checks for each code commit, enhancing feedback and minimizing merge conflicts. These checks encompass tests, code coverage, linting, formatting, and security scrutiny, all aimed at maintaining code quality and application health.

1.4.1 CI Steps

- **Tests**: Use Cargo for executing unit and integration tests to ensure code reliability.

- **Code Coverage**: Tools like **cargo tarpaulin** facilitate the evaluation of test coverage, highlighting areas needing improvement.

- **Linting**: **Clippy** serves as Rust's linter, encouraging best coding practices, which can be integrated into the CI process.

- **Formatting**: **Rustfmt** automates code formatting, ensuring adherence to style guides.

- **Security Vulnerabilities**: **cargo-audit** checks the project's dependencies for known vulnerabilities, enhancing security.

1.4.2 Ready-to-go CI Pipelines

To simplify the CI setup, template configurations for popular CI systems are provided, enabling developers to adjust these templates according to their unique project requirements.

Chapter 2: Building An Email Newsletter

2.1 Our Driving Example

The chapter kicks off with the goal of illustrating the intricacies of constructing cloud-native applications by creating an email newsletter service tailored for independent bloggers, strategically avoiding competition

with larger, established platforms.

2.1.1 Problem-Based Learning

Engaging with real-world problems leads to more effective learning. By focusing on a manageable project like the email newsletter, developers can maintain engagement while addressing relevant challenges.

2.1.2 Course-Correcting

Continuous feedback is vital for progress and improvement. The interactive nature of the project will guide necessary adjustments along the way.

2.2 What Should Our Newsletter Do?

The newsletter should embody core functionalities similar to established services but omit excessive features that do not align with the needs of indie bloggers.

2.2.1 Capturing Requirements: User Stories

Essential features are encapsulated in user stories that outline key functionalities, such as the ability to subscribe to the newsletter, notify subscribers of new content, and facilitate unsubscription.

2.3 Working In Iterations

The focus will be on developing a minimum viable product (MVP) initially, followed by iterative enhancements to expand functionality while upholding quality and thorough testing.

2.3.1 Coming Up

The following chapter will pivot towards the implementation of the subscription functionality. This foundational step will involve significant groundwork, including the setup of web frameworks and infrastructure management necessary for the project.

This approach will not only prepare developers for the technical requirements ahead but also emphasize the importance of laying a strong groundwork in their programming journey.

Chapter 2 Summary: 2 Building An Email Newsletter

Chapter 2: Building An Email Newsletter

2.1 Our Driving Example

This chapter centers on the collaborative development of a cloud-native application, specifically designed to tackle common challenges faced by development teams in creating effective tools. The example chosen to illustrate this process is the development of a simple email newsletter service, which serves as a practical and engaging way to apply theoretical concepts without introducing overwhelming complexity.

2.1.1 Problem-based Learning

The educational approach adopted here emphasizes problem-based learning, where real-world challenges drive the learning process. This method fosters engagement and practicality, allowing learners to connect theory with tangible outcomes. The newsletter project exemplifies this concept, offering a manageable scope to explore essential development techniques.

2.1.2 Course-correcting

An integral part of the learning and development process is the incorporation of feedback. Readers and participants are encouraged to share their insights and suggestions, which serve as a foundation for improving the material and refining the project as it evolves.

2.2 What Should Our Newsletter Do?

The primary goal is to develop a basic email newsletter service tailored for blog authors. Unlike comprehensive platforms such as MailChimp, our focus is on simplicity and functionality that allows authors to communicate effectively with their audience.

2.2.1 Capturing Requirements: User Stories

To define the functionality of the newsletter service, we utilize user stories, which clarify the needs and expectations of potential users. Key user stories include:

1. Blog visitors can easily subscribe to updates.
2. Authors are able to notify subscribers when new content is available.
3. Subscribers have the option to unsubscribe effortlessly.

2.3 Working In Iterations

The development approach emphasizes the importance of iterating over

complete functionalities rather than striving for perfection in individual features from the outset. By focusing on creating a usable product first, we can subsequently enhance aspects such as reliability and user experience through continuous improvements. Each iteration of code must meet production-quality standards to ensure a seamless experience for users.

2.3.1 Coming Up

In the following chapter, we will delve into the mechanics of implementing the subscription functionality, discussing choices of frameworks and initial configurations necessary for the development process.

This chapter lays the foundation for understanding how to build a collaborative project while emphasizing practical learning through solving specific problems. By outlining the project's goals and requirements, it sets the stage for the technical work that will follow.

Chapter 3 Summary: 3 Sign Up A New Subscriber

In Chapter 3, titled "Sign Up A New Subscriber," we delve into the foundational steps necessary for implementing the email newsletter subscription feature for our project. This chapter outlines the backend server's functionalities through the development of a critical POST /subscriptions endpoint while emphasizing strategic considerations in setting up the project.

3.1 Our Strategy

The chapter begins with the establishment of a new project environment, which requires careful planning. Key components include selecting a suitable web framework, developing a comprehensive testing strategy, choosing a database interaction library (or crate), managing database schema migrations, and crafting database queries. Before diving into the complexities of the /subscriptions endpoint, we first create a simple /health_check endpoint to familiarize ourselves with the framework.

3.2 Choosing A Web Framework

For our Rust API, we select actix-web due to its robustness and the strong support it enjoys within the developer community. The framework's compatibility with the Tokio runtime enhances its suitability for production

applications, and resources like documentation and examples will serve as valuable references as we continue to work.

3.3 Our First Endpoint: A Basic Health Check

We proceed to implement a health check endpoint that reliably returns a 200 OK status for GET requests sent to /health_check. This endpoint is crucial for verifying the application's availability and can be integrated with monitoring tools to ensure ongoing reliability.

3.3.1 Wiring Up actix-web

Setting up our actix-web server begins with a basic "Hello World!" example. It’s imperative to install the appropriate dependencies at this stage to avoid compiler errors and ensure the server launches correctly.

3.3.2 Anatomy Of An actix-web Application

The structure of an actix-web application is discussed, focusing on four main components:
- **HttpServer** handles incoming requests and manages connections.

- **App** encapsulates routing, middleware, and request handling.

- **Route** defines how individual routes and their handlers are managed.

- **actix_rt** enables asynchronous programming through non-blocking I/O operations.

3.3.3 Implementing The Health Check Handler

The health check handler is implemented to return a 200 OK response, which is then registered with the server's routing to ensure it functions properly.

3.4 Our First Integration Test

Before moving towards automation, we start by manually testing the health check endpoint using curl. This manual process sets a benchmark for our automated tests.

3.4.1 How Do You Test An Endpoint?

The section discusses the importance of testing APIs to verify they behave as expected and adhere to agreed-upon contracts, thereby preventing breaking changes.

3.4.2 Where Should I Put My Tests?

Tests can be organized in different ways, including embedded modules, external folders, or documentation tests. To enhance clarity, we'll employ an external tests folder for our integration tests.

3.4.3 Changing Our Project Structure For Easier Testing

We refactor the project structure into separate library and binary components. This separation simplifies testing efforts and keeps our main application logic clean while allowing tests to access necessary components easily.

3.5 Implementing Our First Integration Test

We proceed to establish a dedicated integration test for the health check endpoint. This test confirms that the functionality is intact. We further refine our testing strategy by utilizing asynchronous helpers for improved performance.

3.5.1 Polishing

Emphasizing the importance of test independence, we implement strategies such as effective resource pooling and random port allocation to prevent interactions between tests, ensuring reliable outcomes.

3.6 Refocus

With a foothold in our testing framework, we redirect our focus toward the main objective of supporting newsletter subscriptions. Here we outline the backend functionality and database integration required to facilitate this feature.

3.7 Working With HTML Forms

Next, we define the data requirements for subscribers, which include collecting names and email addresses using application/x-www-form-urlencoded POST requests. This specification is crucial for ensuring proper data handling.

3.8 Storing Data: Databases

In evaluating database options, PostgreSQL emerges as the ideal choice for our needs due to its capabilities. We decide to utilize the sqlx crate for safe and efficient database interactions.

3.9 Updating Our Tests

To enhance the integrity of our tests, we introduce isolation techniques that

prevent side effects from leaking between tests. Adopting unique logical databases for each test run plays a key role in maintaining data consistency.

3.10 Summary

In summary, this chapter lays a solid groundwork for building our subscription feature by exploring actix-web, HTML form handling, and database interactions. Additionally, rigorous testing and clean coding practices are emphasized as we integrate more complex functionalities into our application, setting the stage for future developments.

Chapter 4: 4 Telemetry

Chapter 4: Telemetry

In the previous chapter, we successfully established the `POST /subscriptions` endpoint for our email newsletter project, enabling user subscriptions. However, to ensure a robust production launch, we must now focus on enhancing our application's telemetry and observability, crucial for addressing unforeseen issues.

4.1 Unknown Unknowns

While our application includes basic tests, they do not shield us from unexpected failures—termed "unknown unknowns." These encompass unpredictable scenarios like database failures or security threats. To mitigate their impact, we need a deeper understanding of our system's behavior in real-time.

4.2 Observability

Sufficient telemetry data is essential for diagnostics, especially when failures occur outside business hours. It allows us to gain insights to identify root causes without predefined questions. To implement this, we must improve our application instrumentation and utilize effective tools for data analysis.

4.3 Logging

Logs form a vital source of telemetry, providing a narrative of incidents through structured text outputs. They vary based on content and format, making thoughtful logging implementation necessary.

4.3.1 The log Crate

The Rust `log` crate offers macros for logging at varying severity levels (trace, debug, info, warn, error), which necessitates a thoughtful selection of what information to log based on contextual relevance.

4.3.2 actix-web’s Logger Middleware

The `actix_web` framework features a `Logger` middleware that automatically logs incoming requests. To leverage this, we must configure our logging implementation accurately for optimal log capture.

4.3.3 The Facade Pattern

Utilizing a facade pattern with the `log` crate allows us to generate log records without detailing their processing methods. Initializing logging through `set_logger` sets our application's logging behavior.

4.4 Instrumenting POST /subscriptions

To enhance our `POST /subscriptions` handler with robust logging, we must document interactions with external systems and capture critical user details—such as email and name—to facilitate troubleshooting.

4.4.1 Interactions With External Systems

Monitoring network interactions and database queries is crucial for identifying performance issues. By logging these interactions, we can capture necessary details before and after executing queries to better understand potential problems.

4.4.2 Think Like A User

When users, such as "Tom," face subscription issues, our logs should contain enough detail—like the email they attempted to enter—to assist in diagnosing these problems effectively.

4.4.3 Logs Must Be Easy To Correlate

To facilitate tracing issues across concurrent requests, we should integrate a request identifier (UUID) system, allowing us to follow each request's lifecycle through the logs.

4.5 Structured Logging

Traditional logs may struggle to represent intricate workflows, making structured logging via the `tracing` crate a valuable solution. This approach differentiates shared workloads and accurately captures the temporal nature of events.

4.5.1 The tracing Crate

The `tracing` crate improves logging by introducing spans, which encapsulate the start and finish of processes, providing better clarity on their lifecycles.

4.5.2 Migrating From log To tracing

Transitioning from the `log` crate to `tracing` enhances our structured logging by replacing traditional commands with those that facilitate event and span creation.

4.5.3 tracing's Span

With `tracing::info_span`, we can group related log statements under a unified context, aiding in the understanding of application flow when these spans are properly managed.

4.5.4 tracing-futures

Employing `tracing-futures` allows us to instrument asynchronous operations effectively, maintaining context throughout various operations for comprehensive tracking.

4.5.5 tracing's Subscriber

To obtain structured logs, a dedicated subscriber must be implemented, a feature offered by the `tracing-subscriber` crate, providing layered support for enhanced log management.

4.5.6 tracing-subscriber

`tracing-subscriber` includes essential functionalities, like filters and JSON formatting, which markedly improve log clarity and usability.

4.5.7 tracing-bunyan-formatter

Incorporating the `tracing-bunyan-formatter` delivers structured logs in a format that mirrors the familiar `env_logger`, allowing for streamlined future analyses.

4.5.8 tracing-log

To capture all log events comprehensively, we can redirect logs from the `log` crate to our `tracing` subscriber using the `tracing-log`.

4.5.9 Removing Unused Dependencies

Post-migration to `tracing`, it is prudent to remove outdated dependencies (like `log` and `env_logger`) from our `Cargo.toml` file to maintain clean project management.

4.5.10 Cleaning Up Initialization

We further streamline our process by centralizing subscriber setup within a dedicated `telemetry` module, enhancing initialization efficiency.

4.5.11 Logs For Integration Tests

We modify our integration tests to leverage structured logging, ensuring that

every aspect of our system, including logging itself, is thoroughly vetted.

4.5.12 Cleaning Up Instrumentation Code - tracing::instrument

Using the `tracing::instrument` macro simplifies instrumentation by automatically linking input parameters to logging contexts, promoting

Why Bookey is must have App for Book Lovers

30min Content
The deeper and clearer interpretation we provide, the better grasp of each title you have.

Text and Audio format
Absorb knowledge even in fragmented time.

Quiz
Check whether you have mastered what you just learned.

And more
Multiple Voices & fonts, Mind Map, Quotes, IdeaClips...

Free Trial with Bookey

Chapter 5 Summary: 5 Going Live

Chapter 5: Going Live

In this chapter, we focus on deploying a Rust-based newsletter API to production, utilizing Docker containers on DigitalOcean's App Platform. The ultimate goal is to establish a Continuous Deployment (CD) pipeline that supports agile updates and feedback from our users.

5.1 We Must Talk About Deployments

Frequent software deployment is essential for gathering customer feedback and rapidly iterating on our projects. Unfortunately, many resources on deployment lack practical guidance, which can lead to confusion and frustration among developers. Given the complexities of deployment practices, particularly in a book titled *Zero To Production*, understanding deployment mechanisms is crucial for our success.

5.2 Choosing Our Tools

This chapter aims to simulate real-world deployment situations and

implement continuous deployment strategies that will influence our project throughout the text. While crafting the perfect deployment pipeline involves significant complexity, we will focus on pragmatic approaches that balance effectiveness and the developer experience.

5.2.1 Virtualisation: Docker

To prevent compatibility issues between local and production environments, we will employ Docker. This tool allows us to create stable and reproducible environments by packaging our applications and their dependencies together in containers, ensuring consistency across different systems.

5.2.2 Hosting: DigitalOcean

Among various hosting providers, we will choose DigitalOcean's App Platform. Its ease of use and reliability make it an ideal choice for hosting our project, streamlining the deployment process.

5.3 A Dockerfile For Our Application

To deploy our application on DigitalOcean's App Platform, we must create a

Dockerfile that specifies the required environment for our application.

5.3.1 Dockerfiles

A Dockerfile consists of layers assembled from a base image, with specific commands configured to set up the application's environment. For our Rust project, a simple Dockerfile can be constructed as follows:

```dockerfile
FROM rust:1.53.0
WORKDIR /app
COPY . .
RUN cargo build --release
ENTRYPOINT ["./target/release/zero2prod"]
```

5.3.2 Build Context

The build context is vital as it dictates which files are available for the Docker container during the build process. By setting the current directory as the build context, we ensure that we can copy the necessary resources into the container.

5.3.3 Sqlx Offline Mode

During the application's build, we might encounter connectivity issues with the database. SQLx, our asynchronous SQL toolkit, requires compile-time access to the database to ensure query correctness. We can enable SQLx's offline mode to address this by storing metadata for compile-time checks, allowing us to build without live database connectivity.

5.3.4 Running An Image

After successfully building our Docker image, we can run it. However, if the application binds only to the `127.0.0.1` interface, it will reject external requests. Therefore, we need to configure it to listen on `0.0.0.0` to accept connections from any network interface.

5.3.5 Networking

By default, Docker containers do not expose their ports to the host system. To remedy this, we must specify port mapping when running the container (e.g., using `-p 8000:8000`) and ensure our application listens on `0.0.0.0` for external access.

5.3.6 Hierarchical Configuration

To manage different configurations for various environments effectively, we will structure our settings hierarchically. This approach allows for a base configuration with specific overrides for different deployment environments.

5.3.7 Database Connectivity

Facilitating database connections in a production environment requires strategic considerations, such as employing lazy connection pooling and managing sensitive credentials through environment variables. These practices ensure secure and efficient database access.

5.3.8 Optimising Our Docker Image

To enhance efficiency, it is crucial to reduce the size of our Docker image and improve build times. We can achieve this through multi-stage builds, where we create separate environments for building and running the application, optimizing storage and speed.

5.4 Deploy To DigitalOcean Apps Platform

With our Docker image complete, we will proceed with deploying to DigitalOcean's App Platform. This involves creating an application specification file, linking it to our GitHub repository, and setting environment variables for database configuration. Once deployed, we will test the API endpoints to confirm that everything operates as intended.

In conclusion, this chapter equips us with the knowledge and tools required to deploy our Rust application successfully on DigitalOcean, while also effectively managing deployments through code. By embracing these practices, we set a solid foundation for ongoing application development and improvement.

Chapter 6 Summary: 6 Reject Invalid Subscribers #1

Chapter 6: Reject Invalid Subscribers

In this chapter, we navigate the complexities of validating user input for our newsletter API. Although our API is operational with monitoring tools in place, we've identified weaknesses in input validation, particularly at the POST /subscriptions endpoint, where it currently only checks for the presence of names and emails. This oversight could lead to erroneous entries in our database, particularly as tests have shown that invalid inputs still yield a 200 OK response.

6.1 Requirements

Input validation faces a dual challenge: **Domain Constraints** and **Security C onstraints**. Validating names is inherently complex due to their diversity; thus, we only require names to be non-empty to simplify the process. In terms of security, we recognize that our forms are vulnerable to various attacks, including denial of service and phishing. While we won't tackle every potential threat within our validation logic, we adopt a layered security approach to mitigate risks across different levels.

6.2 First Implementation

To address the lax validation for subscriber names, we implement a new

function that enforces checks on names to ensure they are non-empty, do not exceed 256 characters, and are free of problematic characters.

6.3 Validation Is A Leaky Cauldron

We acknowledge that relying on a single validation point is inefficient, particularly as our project scales. Ensuring validation at every function call can lead to overwhelming complexity.

6.4 Type-Driven Development

To enforce strong validation rules, we introduce a `SubscriberName` struct, utilizing Rust's type system. This allows us to guarantee that only valid names can be processed at compile time, preventing invalid inputs from ever reaching runtime.

6.5 Ownership Meets Invariants

With the new definition of `insert_subscriber`, we enforce that only valid subscriber names are inserted. We also create methods for safe access to the `SubscriberName`'s inner string, exploring Rust's `AsRef` trait for improved access patterns.

6.6 Panics

Currently, validation failures result in application panics, which is unacceptable in a production environment. We refactor our code to use Result types instead, facilitating graceful error handling.

6.7 Error as Values - Result

The Result type allows us to manage errors systematically, eliminating the chaos of exceptions. We adjust our validation methods to return Results, permitting efficient error management without crashing the API.

6.8 Insightful Assertion Errors: claim

To enhance debugging, we implement the ‘claim’ crate, which clarifies assertion failures during testing, making it easier for developers to spot issues.

6.9 Unit Tests

We create unit tests for the `SubscriberName` struct to verify that our validation works effectively. The challenge of managing panic situations is resolved by ensuring our validation methods return Results instead.

6.10 Handling a Result

Next, we update our API handler to correctly manage the results produced by the `SubscriberName::parse` function, ensuring that invalid submissions result in a proper 400 Bad Request response.

6.11 The Email Format

Email validation introduces additional challenges. We choose to utilize existing libraries, such as the validator crate, to assist in this validation

process, allowing us to streamline our approach.

6.12 The SubscriberEmail Type

We introduce a `SubscriberEmail` type that applies similar validation principles as the `SubscriberName`, creating necessary routines for processing email inputs.

6.13 Property-Based Testing

To further strengthen our email validation, we implement property-based testing, which evaluates the general attributes of valid email addresses rather than validating against specific examples.

6.14 Payload Validation

As we integrate `SubscriberEmail` into our API, we ensure that all user inputs undergo rigorous validation, appropriately responding to either valid or invalid submissions.

6.15 Summary

This chapter underscores the criticality of comprehensive user input validation and the advantages of type-driven development for maintaining data integrity. We also introduce the need for confirmation emails to verify addresses, paving the way for this feature in the subsequent chapter.

Chapter 7 Summary: 7 Reject Invalid Subscribers #2

Chapter 7: Reject Invalid Subscribers

In the ongoing development of a robust subscriber management system, Chapter 7 focuses on confirming the validity of new subscribers through a systematic approach to email verification. The objective is to enhance the integrity of the subscriber list while ensuring compliance with legal standards.

7.1 Confirmation Emails

Previously discussed was the basic validation of new subscribers' email addresses, ensuring they met syntactic standards. However, validating the real existence of these addresses is crucial. This is achieved through the sending of confirmation emails.

7.1.1 Subscriber Consent

Seeking explicit consent from subscribers is paramount, particularly due to regulations like the General Data Protection Regulation (GDPR) in Europe, which safeguards against unsolicited subscriptions. The confirmation email serves as a protective measure to confirm the subscriber's intent.

7.1.2 The Confirmation User Journey

Upon subscription, users receive a confirmation email with a unique link. When clicked, this link directs them to a success page, confirming their subscription. The backend processes the request by marking it with a pending status, generating a unique token, and later updating the user's status to active upon link verification.

7.1.3 The Implementation Strategy

The implementation will unfold in three phases:
1. Developing a module dedicated to email sending.
2. Modifying the existing subscription handler to incorporate confirmation procedures.
3. Establishing a new handler specifically for managing confirmation responses.

7.2 EmailClient: Our Email Delivery Component

7.2.1 How To Send An Email

Email delivery typically leverages SMTP; however, utilizing a third-party service with a REST API simplifies the process. To facilitate this, a REST

client will be created for seamless communication with the chosen email delivery service.

7.2.1.1 Choosing An Email API

Several reliable email service providers exist, including AWS SES, SendGrid, and Postmark. For its simplicity and user-friendly onboarding experience, Postmark has been selected for this project.

7.2.1.2 The Email Client Interface

The email client will feature a `send_email` method that requires the recipient's email, subject line, and both HTML and plain text content for the email.

7.2.2 How To Write A REST Client Using reqwest

To facilitate HTTP requests, the `reqwest` library will be employed, valued for its support of asynchronous operations and user-friendliness.

7.2.2.1 reqwest::Client

An instance of `reqwest::Client` will be created to execute requests efficiently. The EmailClient struct will contain this client alongside

configuration details such as the base URL and sender information.

7.2.2.2 Connection Pooling

Connection pooling will be utilized to enhance performance by reusing established connections rather than creating new ones for each email-sending request.

7.2.2.3 Reusing reqwest::Client in actix-web

To optimize performance, the `reqwest::Client` will be stored within the application context, ensuring it is readily available during request handling.

7.2.2.4 Configuring Our EmailClient

To instantiate an `EmailClient`, specific configurations—including the email service's base URL and the sender's email—will be integrated into the settings structure.

7.2.3 How To Test A REST Client

Unit testing will commence with the EmailClient to verify its functionality in isolation. Mocking an HTTP server will facilitate testing the email sending function without dispatching actual emails.

7.2.3.1 HTTP Mocking With wiremock

The `wiremock` library will be utilized to simulate requests to the email API, ensuring the email client behaves as expected.

7.2.3.2 wiremock::MockServer

A mock server will be set up to listen for incoming requests and check interactions against defined expectations.

7.2.3.3 wiremock::Mock

Mocks will be tailored to define conditions for request matching, ensuring the email client requests adhere to specified parameters.

7.2.3.4 The Intent Of A Test Should Be Clear

A crucial testing principle is that the specific inputs used are less significant than the overall behavior observed under varying conditions.

7.2.3.5 Mock Expectations

Tests will conclude by verifying that the anticipated number of requests was

dispatched, with failure resulting from discrepancies between actual and expected outcomes.

7.2.4 First Sketch Of EmailClient::send_email

The development of the email-sending function will progress in accordance with the REST API documentation from Postmark, ensuring accurate implementation and functionality.

Through these systematic processes and strategic implementations, Chapter 7 establishes a foundation for maintaining a high-quality subscriber list, crucial for effective digital communication.

Chapter 8: 8 Error Handling

Chapter 8: Error Handling

In the realm of software development, managing errors effectively is crucial, especially when sending confirmation emails involves various operations such as user input validation, email dispatch, and database queries that can all fail. This chapter explores error handling patterns in Rust, with a focus on their implications for application architecture, the purpose of errors, suitable libraries for error management, and the characteristics of effective error representation.

8.1 What Is The Purpose Of Errors?

Through the example of an asynchronous function called `store_token`, the chapter introduces Rust's `Result` type, which informs the caller whether an operation succeeded or failed. Errors serve two primary purposes:

1. **Control Flow:** They allow callers to decide how to proceed following an operation's outcome.
2. **Reporting:** They provide context that aids in diagnosing issues.

Two error categories are distinguished:

- **Internal Errors:** These occur within the application.

- **Errors At The Edge:** These are related to API interactions involving requests and responses.

An effective error type is one that communicates failures clearly, benefiting both machine interactions (via types and codes) and human understanding (through descriptive messages).

8.2 Error Reporting For Operators

The chapter emphasizes the significant role of logging in operational error reporting. It discusses the necessity of crafting useful log messages and provides a test case concerning database errors to illustrate how clarity in logs can be achieved. The guidelines stress that errors should be tracked at the point of processing rather than when they are passed upstream, ensuring clearer diagnostics.

8.2.1 Keeping Track Of The Error Root Cause

The discourse here shifts to error propagation and how to enhance errors with additional context, thereby simplifying logging and debugging tasks. Special attention is given to leveraging `actix_web::Error` for tailored error

management in web applications, which further clarifies error sources and simplifies problem resolution.

8.3 Errors For Control Flow

A proper separation of concerns is essential for effective error handling, exemplified through a custom error type, `SubscribeError`, which encapsulates HTTP-related logic. By introducing an enum structure for categorizing different errors, the chapter demonstrates how clearer error handling can be achieved. The section also discusses boilerplate code commonly associated with error handling and suggests the use of macros from libraries such as `thiserror` to simplify tasks like implementing `Debug` and `Display` traits, thus reducing code complexity.

8.4 Avoid “Ball Of Mud” Error Enums

The chapter warns against the dangers of blending unrelated concerns into a single error type, which can lead to a "ball of mud" scenario. Instead, it advocates for a clearer, more abstract error type that organizes errors into logical categories such as `ValidationError` and `UnexpectedError`, simplifying error management.

8.5 Who Should Log Errors?

The chapter further explores the process of logging, arguing that errors should generally be logged at the point of handling rather than during propagation. This practice helps to mitigate duplicate log entries and enhances clarity for operators. The emphasis is on crafting log entries that provide substantial context regarding the operation that failed and the nature

App Store
Editors' Choice

22k 5 star review

Positive feedback

Sara Scholz

tes after each book summary
erstanding but also make the
and engaging. Bookey has
ding for me.

Fantastic!!! Masood El Toure

I'm amazed by the variety of books and languages Bookey supports. It's not just an app, it's a gateway to global knowledge. Plus, earning points for charity is a big plus!

Fi
Ab
bo
to
m

José Botín

ding habit
's design
ual growth

Love it! Wonnie Tappkx

Bookey offers me time to go through the important parts of a book. It also gives me enough idea whether or not I should purchase the whole book version or not! It is easy to use!

Time saver!

Bookey is my go-to app for
summaries are concise, ins
curated. It's like having acc
right at my fingertips!

Awesome app! Rahul Malviya

I love audiobooks but don't always have time to listen to the entire book! bookey allows me to get a summary of the highlights of the book I'm interested in!!! What a great concept !!!highly recommended!

Beautiful App Alex Walk

This app is a lifesaver for book lovers with busy schedules. The summaries are spot on, and the mind maps help reinforce wh I've learned. Highly recommend!

Free Trial with Bookey

Chapter 9 Summary: 9 Naive Newsletter Delivery

Chapter 9: Naive Newsletter Delivery

In this chapter, we embark on developing a foundational newsletter delivery system for our previously established blog platform. This step is essential for enhancing our understanding of email functionalities and preparing for more complex elements like user authentication and fault tolerance in the future.

9.1 User Stories Are Not Set In Stone

The initial user story described
the need for a blog author to send emails to all subscribers to notify them of new content. However, after refinements to our subscriber model—which now distinguishes between confirmed and unconfirmed subscribers—we adjusted the user story, emphasizing that only confirmed subscribers should receive newsletters. This refinement highlights the importance of clear and adaptable user requirements in software development.

9.2 Do Not Spam Unconfirmed Subscribers

The first priority is ensuring that unconfirmed subscribers do not receive newsletters. To achieve this, we implement integration tests using Postmark as our email service provider. These tests verify that no emails are dispatched if all subscribers lack confirmation.

9.2.1 Set Up State Using The Public API

We adopt a black-box testing strategy, setting up our tests on a fresh application instance with an empty database by invoking its public API. This ensures a reliable testing environment.

9.2.2 Scoped Mocks

While creating unconfirmed subscribers, we utilize scoped mocks, which isolate our email service's behavior during tests, preventing interference between different test scenarios.

9.2.3 Green Test

To facilitate passing our initial tests, we establish a temporary endpoint for sending newsletters, allowing us to simulate the newsletter delivery process.

9.3 All Confirmed Subscribers Receive New Issues

Next, we write integration tests to confirm that our system sends newsletters to all confirmed subscribers. To streamline development and avoid code duplication, we develop helper functions dedicated to creating these confirmed subscribers.

9.4 Implementation Strategy

Our implementation will extract newsletter details from incoming requests, retrieve confirmed subscribers from the database, and utilize Postmark to

send the emails.

9.5 Body Schema

To structure our newsletter delivery, we define the title and content format using appropriate data structures that implement `serde::Deserialize`, facilitating the parsing of JSON input for the newsletter.

9.5.1 Test Invalid Inputs

We will enhance our tests to ensure that any inappropriate input data triggers a 400 Bad Request response from our `POST /newsletters` endpoint. This helps to enforce input validation and robust error handling.

9.6 Fetch Confirmed Subscribers List

An SQL query will be crafted to efficiently obtain a list of all confirmed subscribers by only retrieving essential fields, optimizing our data retrieval process.

9.7 Send Newsletter Emails

The implementation phase focuses on sending formatted emails to the subscribers identified from the database. Ensuring that the emails adhere to the necessary format is crucial for successful delivery.

9.8 Validation of Stored Data

As we retrieve verified email addresses from the database, we prioritize

validation. This process involves filtering out any invalid email formats and issuing appropriate warnings to maintain data integrity.

9.8.1 Responsibility Boundaries

Decisions regarding the handling of invalid email addresses should be made higher in the workflow, particularly in the main email-sending function, rather than merely at the data retrieval level. This approach strengthens the overall reliability of our email sending process.

9.9 Limitations Of The Naive Approach

While our initial implementation successfully passes tests, it exhibits several limitations. Notably, there is a lack of security measures for the API endpoint, no capability for saving draft newsletters, performance challenges due to synchronous email sending, and a failure to incorporate fault tolerance and retry capabilities.

9.10 Summary

In conclusion, this chapter successfully establishes a working but primitive prototype of our newsletter delivery system that meets basic functional requirements. However, for it to be production-ready, significant enhancements focusing on security, performance, and fault tolerance are urgently needed in the forthcoming chapters.

Chapter 10 Summary: 10 Securing Our API

Chapter 10: Securing Our API

In the previous chapter, we introduced a new endpoint, POST /newsletters, which enables the sending of newsletters to subscribers. However, this feature lacks essential security, allowing unauthorized access and potential misuse. Chapter 10 aims to enhance the API's security by implementing robust authentication and authorization measures.

10.1 Authentication

Authentication ensures that only verified users can access the POST /newsletters endpoint. This process includes several methods categorized into three main groups:

- **Something they know**: This includes passwords or answers to security questions.
- **Something they have**: Examples are authenticator apps or mobile devices.
- **Something they are**: This involves biometrics, such as fingerprints.

However, each method presents certain weaknesses that must be addressed.

10.1.1 Drawbacks

1. **Something They Know**: Users often struggle to manage long, unique passwords without reusing them.
2. **Something They Have**: Physical devices can be lost or compromised.

3. **Something They Are**: Biometrics are immutable and can be susceptible to forgery.

10.1.2 Multi-factor Authentication (MFA)

To counter the weaknesses of single-factor authentication, Multi-factor Authentication (MFA) is recommended. This approach requires users to provide at least two different types of credentials, enhancing security.

10.2 Password-based Authentication

This section elaborates on implementing password-based authentication, particularly emphasizing the ‘Basic’ Authentication Scheme that employs base64 encoding for credentials in the Authorization header.

10.2.1 Basic Authentication

The API is designed to expect a structured Authorization header. Requests lacking this header or containing invalid credentials will be rejected, resulting in a 401 Unauthorized status.

10.2.1.1 Extracting Credentials

Upon receiving a request, the system extracts the username and password using base64 decoding, rejecting requests that do not have a properly formatted Authorization header.

10.2.2 Password Verification - Naive Approach

Initially, a simplistic password check is demonstrated using hardcoded credentials, but this method is inadequate. The chapter suggests transitioning to a database for user management, where usernames and hashed passwords are securely stored.

10.2.3 Password Storage

Storing plaintext passwords poses significant risks, hence emphasizing the necessity of hashing. The chapter recommends Argon2id, a modern hashing standard endorsed by OWASP, for secure password storage.

10.2.3.3 Preimage Attack

An explanation of preimage attacks highlights that their difficulty is directly linked to the length of the hash, underscoring the importance of using sufficiently long hashes.

10.2.3.5 Dictionary Attack

The chapter discusses the threat of dictionary attacks, where common passwords are exploited. It stresses the importance of utilizing slow hashing algorithms to mitigate this type of vulnerability.

10.2.4 Do Not Block The Async Executor

To maintain efficiency during password validation, the chapter cautions against blocking the async executor. Instead, it suggests employing `tokio::task::spawn_blocking` for CPU-intensive operations.

10.2.5 User Enumeration

Concerns are raised about user enumeration risks, particularly due to timing attacks. Strategies to reduce these vulnerabilities include randomizing response times and limiting failed login attempts to thwart brute-force attacks.

10.3 Is it Safe?

This section evaluates the security measures implemented, particularly the necessity for Transport Layer Security (TLS) to protect credentials during transmission and the essential features of a password reset mechanism.

10.3.1 Transport Layer Security (TLS)

The chapter underscores the importance of employing TLS when using basic authentication, as it safeguards against potential eavesdropping attacks.

10.4 What Should We Do Next

Concluding the chapter, it is stated that Basic Authentication is primarily suitable for machine-to-machine communications. The chapter recommends exploring session-based authentication for user engagements through web browsers. Additionally, it encourages looking into OAuth2 and OpenID Connect for federated login solutions in future discussions.

This chapter effectively establishes a foundation for securing the API's new capabilities and emphasizes the importance of thoughtful security design in software development.