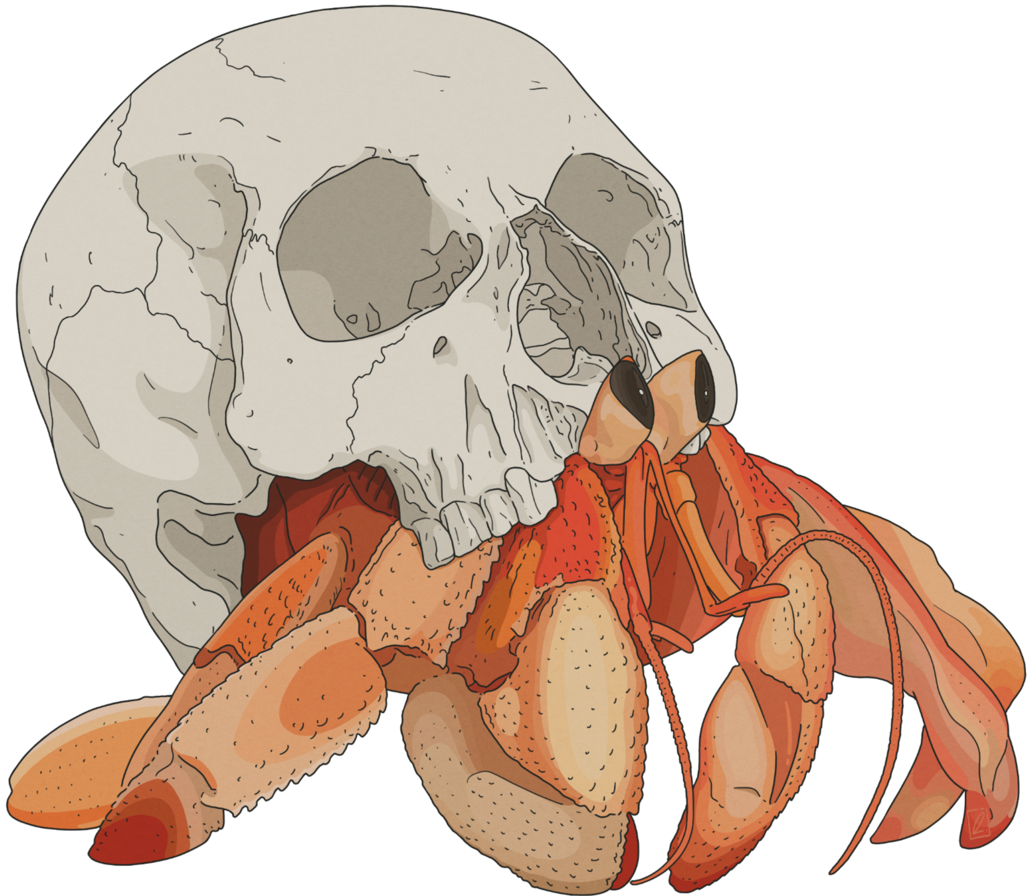
ZERO TO PRODUCTION IN RUST

AN OPINIONATED INTRODUCTION TO BACKEND DEVELOPMENT



LUCA PALMIERI

A Federica,
il mio porto sicuro.

Contents

Foreword **xiii**

Preface **xv**
What Is This Book About . . . xv
Cloud-native applications . . . xv
Working in a team . . . xvi
Who Is This Book For . . . xvii

1 Getting Started **1**
1.1 Installing The Rust Toolchain . . . 1
1.1.1 Compilation Targets . . . 1
1.1.2 Release Channels . . . 2
1.1.3 What Toolchains Do We Need? . . . 2
1.2 Project Setup . . . 3
1.3 IDEs . . . 4
1.3.1 Rust-analyzer . . . 4
1.3.2 RustRover . . . 4
1.3.3 What Should I Use? . . . 4
1.4 Inner Development Loop . . . 5
1.4.1 Faster Linking . . . 5
1.4.2 `cargo-watch` . . . 6
1.5 Continuous Integration . . . 7
1.5.1 CI Steps . . . 8
1.5.2 Ready-to-go CI Pipelines . . . 11

2 Building An Email Newsletter **13**
2.1 Our Driving Example . . . 13
2.1.1 Problem-based Learning . . . 13
2.2 What Should Our Newsletter Do? . . . 14
2.2.1 Capturing Requirements: User Stories . . . 14
2.3 Working In Iterations . . . 15
2.3.1 Coming Up . . . 15
2.4 Checking Your Progress . . . 16

3 Sign Up A New Subscriber **17**
3.1 Our Strategy . . . 17

3.2 Choosing A Web Framework 18
3.3 Our First Endpoint: A Basic Health Check 18
3.3.1 Wiring Up `actix-web` 18
3.3.2 Anatomy Of An `actix-web` Application 20
3.3.3 Implementing The Health Check Handler 25
3.4 Our First Integration Test 28
3.4.1 How Do You Test An Endpoint? 28
3.4.2 Where Should I Put My Tests? 30
3.4.3 Changing Our Project Structure For Easier Testing 31
3.5 Implementing Our First Integration Test 35
3.5.1 Polishing 39
3.6 Refocus 43
3.7 Working With HTML Forms 44
3.7.1 Refining Our Requirements 44
3.7.2 Capturing Our Requirements As Tests 44
3.7.3 Parsing Form Data From A `POST` Request 47
3.8 Storing Data: Databases 56
3.8.1 Choosing A Database 56
3.8.2 Choosing A Database Crate 57
3.8.3 Integration Testing With Side-effects 58
3.8.4 Database Setup 59
3.8.5 Writing Our First Query 66
3.9 Persisting A New Subscriber 74
3.9.1 Application State In `actix-web` 75
3.9.2 `actix-web` Workers 77
3.9.3 The `Data` Extractor 79
3.9.4 The `INSERT` Query 80
3.10 Updating Our Tests 84
3.10.1 Test Isolation 88
3.11 Summary 91

4 Telemetry **93**
4.1 Unknown Unknowns 93
4.2 Observability 94
4.3 Logging 95
4.3.1 The `log` Crate 96
4.3.2 `actix-web`'s `Logger` Middleware 97
4.3.3 The Facade Pattern 97
4.4 Instrumenting POST /subscriptions 100
4.4.1 Interactions With External Systems 101
4.4.2 Think Like A User 102
4.4.3 Logs Must Be Easy To Correlate 104
4.5 Structured Logging 106
4.5.1 The `tracing` Crate 107
4.5.2 Migrating From `log` To `tracing` 107

- 4.5.3 `tracing`'s `Span` . . . 108
- 4.5.4 Instrumenting Futures . . . 111
- 4.5.5 `tracing`'s `Subscriber` . . . 113
- 4.5.6 `tracing-subscriber` . . . 114
- 4.5.7 `tracing-bunyan-formatter` . . . 114
- 4.5.8 `tracing-log` . . . 117
- 4.5.9 Removing Unused Dependencies . . . 118
- 4.5.10 Cleaning Up Initialisation . . . 118
- 4.5.11 Logs For Integration Tests . . . 122
- 4.5.12 Cleaning Up Instrumentation Code - `tracing::instrument` . . . 126
- 4.5.13 Protect Your Secrets - `secrecy` . . . 130
- 4.5.14 Request Id . . . 133
- 4.5.15 Leveraging The `tracing` Ecosystem . . . 136
- 4.6 Summary . . . 136

5 Going Live **137**
- 5.1 We Must Talk About Deployments . . . 137
- 5.2 Choosing Our Tools . . . 138
  - 5.2.1 Virtualisation: Docker . . . 138
  - 5.2.2 Hosting: DigitalOcean . . . 139
- 5.3 A Dockerfile For Our Application . . . 139
  - 5.3.1 Dockerfiles . . . 139
  - 5.3.2 Build Context . . . 140
  - 5.3.3 Sqlx Offline Mode . . . 141
  - 5.3.4 Running An Image . . . 143
  - 5.3.5 Networking . . . 145
  - 5.3.6 Hierarchical Configuration . . . 146
  - 5.3.7 Database Connectivity . . . 152
  - 5.3.8 Optimising Our Docker Image . . . 152
- 5.4 Deploy To DigitalOcean Apps Platform . . . 158
  - 5.4.1 Setup . . . 158
  - 5.4.2 App Specification . . . 158
  - 5.4.3 How To Inject Secrets Using Environment Variables . . . 162
  - 5.4.4 Connecting To Digital Ocean's Postgres Instance . . . 164
  - 5.4.5 Environment Variables In The App Spec . . . 167
  - 5.4.6 One Last Push . . . 168

6 Reject Invalid Subscribers #1 **169**
- 6.1 Requirements . . . 170
  - 6.1.1 Domain Constraints . . . 170
  - 6.1.2 Security Constraints . . . 170
- 6.2 First Implementation . . . 172
- 6.3 Validation Is A Leaky Cauldron . . . 174
- 6.4 Type-Driven Development . . . 175
- 6.5 Ownership Meets Invariants . . . 179

6.5.1 AsRef . . . 182
6.6 Panics . . . 185
6.7 Error As Values - Result . . . 187
6.7.1 Converting parse To Return Result . . . 187
6.8 Insightful Assertion Errors: claims . . . 190
6.9 Unit Tests . . . 191
6.10 Handling A Result . . . 194
6.10.1 match . . . 194
6.10.2 The ? Operator . . . 195
6.10.3 400 Bad Request . . . 196
6.11 The Email Format . . . 196
6.12 The SubscriberEmail Type . . . 197
6.12.1 Breaking The Domain Sub-Module . . . 197
6.12.2 Skeleton Of A New Type . . . 199
6.13 Property-based Testing . . . 201
6.13.1 How To Generate Random Test Data With fake . . . 202
6.13.2 quickcheck Vs proptest . . . 203
6.13.3 Getting Started With quickcheck . . . 203
6.13.4 Implementing The Arbitrary Trait . . . 204
6.14 Payload Validation . . . 206
6.14.1 Refactoring With TryFrom . . . 210
6.15 Summary . . . 213

7 Reject Invalid Subscribers #2 215
7.1 Confirmation Emails . . . 215
7.1.1 Subscriber Consent . . . 215
7.1.2 The Confirmation User Journey . . . 216
7.1.3 The Implementation Strategy . . . 217
7.2 EmailClient, Our Email Delivery Component . . . 217
7.2.1 How To Send An Email . . . 217
7.2.2 How To Write A REST Client Using reqwest . . . 220
7.2.3 How To Test A REST Client . . . 228
7.2.4 First Sketch Of EmailClient::send_email . . . 234
7.2.5 Tightening Our Happy Path Test . . . 242
7.2.6 Dealing With Failures . . . 251
7.3 Skeleton And Principles For A Maintainable Test Suite . . . 261
7.3.1 Why Do We Write Tests? . . . 261
7.3.2 Why Don't We Write Tests? . . . 262
7.3.3 Test Code Is Still Code . . . 262
7.3.4 Our Test Suite . . . 263
7.3.5 Test Discovery . . . 264
7.3.6 One Test File, One Crate . . . 264
7.3.7 Sharing Test Helpers . . . 265
7.3.8 Sharing Startup Logic . . . 269
7.3.9 Build An API Client . . . 278

7.3.10 Summary . . . 282
7.4 Refocus . . . 282
7.5 Zero Downtime Deployments . . . 283
7.5.1 Reliability . . . 283
7.5.2 Deployment Strategies . . . 283
7.6 Database Migrations . . . 286
7.6.1 State Is Kept Outside The Application . . . 286
7.6.2 Deployments And Migrations . . . 287
7.6.3 Multi-step Migrations . . . 288
7.6.4 A New Mandatory Column . . . 288
7.6.5 A New Table . . . 290
7.7 Sending A Confirmation Email . . . 291
7.7.1 A Static Email . . . 291
7.7.2 A Static Confirmation Link . . . 296
7.7.3 Pending Confirmation . . . 300
7.7.4 Skeleton of `GET /subscriptions/confirm` . . . 304
7.7.5 Connecting The Dots . . . 307
7.7.6 Subscription Tokens . . . 317
7.8 Database Transactions . . . 325
7.8.1 All Or Nothing . . . 325
7.8.2 Transactions In Postgres . . . 326
7.8.3 Transactions In Sqlx . . . 326
7.9 Summary . . . 331

8 Error Handling **333**
8.1 What Is The Purpose Of Errors? . . . 333
8.1.1 Internal Errors . . . 334
8.1.2 Errors At The Edge . . . 336
8.1.3 Summary . . . 338
8.2 Error Reporting For Operators . . . 339
8.2.1 Keeping Track Of The Error Root Cause . . . 342
8.2.2 The `Error` Trait . . . 348
8.3 Errors For Control Flow . . . 352
8.3.1 Layering . . . 352
8.3.2 Modelling Errors as Enums . . . 353
8.3.3 The Error Type Is Not Enough . . . 355
8.3.4 Removing The Boilerplate With `thiserror` . . . 359
8.4 Avoid "Ball Of Mud" Error Enums . . . 361
8.4.1 Using `anyhow` As Opaque Error Type . . . 366
8.4.2 `anyhow` Or `thiserror`? . . . 369
8.5 Who Should Log Errors? . . . 369
8.6 Summary . . . 371

9 Naive Newsletter Delivery **373**
9.1 User Stories Are Not Set In Stone . . . 373

9.2 Do Not Spam Unconfirmed Subscribers . . . 374
9.2.1 Set Up State Using The Public API . . . 376
9.2.2 Scoped Mocks . . . 376
9.2.3 Green Test . . . 377
9.3 All Confirmed Subscribers Receive New Issues . . . 378
9.3.1 Composing Test Helpers . . . 378
9.4 Implementation Strategy . . . 380
9.5 Body Schema . . . 381
9.5.1 Test Invalid Inputs . . . 382
9.6 Fetch Confirmed Subscribers List . . . 384
9.7 Send Newsletter Emails . . . 387
9.7.1 `context` Vs `with_context` . . . 388
9.8 Validation Of Stored Data . . . 389
9.8.1 Responsibility Boundaries . . . 393
9.8.2 Follow The Compiler . . . 395
9.8.3 Remove Some Boilerplate . . . 396
9.9 Limitations Of The Naive Approach . . . 398
9.10 Summary . . . 399

10 Securing Our API 401
10.1 Authentication . . . 401
10.1.1 Drawbacks . . . 402
10.1.2 Multi-factor Authentication . . . 402
10.2 Password-based Authentication . . . 402
10.2.1 Basic Authentication . . . 403
10.2.2 Password Verification - Naive Approach . . . 409
10.2.3 Password Storage . . . 412
10.2.4 Do Not Block The Async Executor . . . 430
10.2.5 User Enumeration . . . 438
10.3 Is it safe? . . . 442
10.3.1 Transport Layer Security (TLS) . . . 442
10.3.2 Password Reset . . . 443
10.3.3 Interaction Types . . . 443
10.3.4 Machine To Machine . . . 443
10.3.5 Person Via Browser . . . 444
10.3.6 Machine to machine, on behalf of a person . . . 445
10.4 Interlude: Next Steps . . . 445
10.5 Login Forms . . . 445
10.5.1 Serving HTML Pages . . . 445
10.6 Login . . . 448
10.6.1 HTML Forms . . . 449
10.6.2 Redirect On Success . . . 452
10.6.3 Processing Form Data . . . 453
10.6.4 Contextual Errors . . . 462
10.7 Sessions . . . 496

10.7.1 Session-based Authentication 496
10.7.2 Session Store 497
10.7.3 Choosing A Session Store 497
10.7.4 `actix-session` 498
10.7.5 Admin Dashboard 502
10.8 Seed Users 514
10.8.1 Database Migration 515
10.8.2 Password Reset 516
10.9 Refactoring 534
10.9.1 How To Write An `actix-web` Middleware 536
10.10 Summary 542

11 Fault-tolerant Workflows 545
11.1 `POST /admin/newsletters` - A Refresher 545
11.2 Our Goal 547
11.3 Failure Modes 547
11.3.1 Invalid Inputs 547
11.3.2 Network I/O 548
11.3.3 Application Crashes 549
11.3.4 Author Actions 549
11.4 Idempotency: An Introduction 549
11.4.1 Idempotency In Action: Payments 550
11.4.2 Idempotency Keys 551
11.4.3 Concurrent Requests 552
11.5 Requirements As Tests #1 552
11.6 Implementation Strategies 554
11.6.1 Stateful Idempotency: Save And Replay 554
11.6.2 Stateless Idempotency: Deterministic Key Generation 554
11.6.3 Time Is a Tricky Beast 554
11.6.4 Making A Choice 555
11.7 Idempotency Store 555
11.7.1 Which Database Should We Use? 555
11.7.2 Schema 556
11.8 Save And Replay 557
11.8.1 Read Idempotency Key 557
11.8.2 Retrieve Saved Responses 562
11.8.3 Save Responses 565
11.9 Concurrent Requests 573
11.9.1 Requirements As Tests #2 573
11.9.2 Synchronization 575
11.10 Dealing With Errors 582
11.10.1 Distributed Transactions 584
11.10.2 Backward Recovery 585
11.10.3 Forward Recovery 585
11.10.4 Asynchronous Processing 586

11.11 Epilogue 604

Foreword

When you read these lines, Rust has achieved its biggest goal: make an offer to programmers to write their production systems in a different language. By the end of the book, it is still your choice to follow that path, but you have all you need to consider the offer. I've been part of the growth process of two widely different languages: Ruby and Rust - by programming them, but also by running events, being part of their project management and running business around them. Through that, I had the privilege of being in touch with many of the creators of those languages and consider some of them friends. Rust has been my one chance in life to see and help a language grow from the experimental stage to adoption in the industry.

I'll let you in on a secret I learned along the way: programming languages are not adopted because of a feature checklist. It's a complex interplay between good technology, the ability to speak about it and finding enough people willing to take long bets. When I write these lines, over 5000 people have contributed to the Rust project, often for free, in their spare time - because they believe in that bet. But you don't have to contribute to the compiler or be recorded in a git log to contribute to Rust. Luca's book is such a contribution: it gives newcomers a perspective on Rust and promotes the good work of those many people.

Rust was never intended to be a research platform - it was always meant as a programming language solving real, tangible issues in large codebases. It is no surprise that it comes out of an organization that maintains a very large and complex codebase - Mozilla, creators of Firefox. When I joined Rust, it was just ambition - but the ambition was to industrialize research to make the software of tomorrow better. With all of its theoretical concepts, linear typing, region based memory management, the programming language was always meant for everyone. This reflects in its lingo: Rust uses accessible names like "Ownership" and "Borrowing" for the concepts I just mentioned. Rust is an industry language, through and through.

And that reflects in its proponents: I've known Luca for years as a community member who knows a ton about Rust. But his deeper interest lies in convincing people that Rust is worth a try by addressing their needs. The title and structure of this book reflects one of the core values of Rust: to find its worth in writing production software that is solid and works. Rust shows its strength in the care and knowledge that went into it to write stable software productively. Such an experience is best found with a guide and Luca is one of the best guides you can find around Rust.

Rust doesn't solve all of your problems, but it has made an effort to eliminate whole categories of mistakes. There's the view out there that safety features in languages are there because of the incompetence of programmers. I don't subscribe to this view. Emily Dunham, captured it well in her RustConf 2017 keynote: "safe code allows you to take better risks". Much of the magic of the Rust community lies in this positive view of its users: whether you are a newcomer or an experienced developer, we trust your experience and your decision-making. In this book, Luca offers a lot of new knowledge that can be applied even outside of

Rust, well explained in the context of daily software praxis. I wish you a great time reading, learning and contemplating.

Florian Gilcher,
Management Director of Ferrous Systems and
Co-Founder of the Rust Foundation

Preface

What Is This Book About

The world of backend development is **vast**.

The context you operate into has a huge impact on the optimal tools and practices to tackle the problem you are working on.

For example, trunk-based development works extremely well to write software that is continuously deployed in a Cloud environment.
The very same approach might fit poorly the business model and the challenges faced by a team that sells software that is hosted and run on-premise by their customers - they are more likely to benefit from a Gitflow approach.
If you are working alone, you can just push straight to `main`.

There are few absolutes in the field of software development and I feel it's beneficial to clarify your point of view when evaluating the pros and cons of any technique or approach.

> *Zero To Production* will focus on the challenges of writing Cloud-native applications in a team of four or five engineers with different levels of experience and proficiency.

Cloud-native applications

Defining what *Cloud-native application* means is, by itself, a topic for a whole new book[1]. Instead of prescribing what Cloud-native applications should *look like*, we can lay down what we expect them to *do*.
Paraphrasing Cornelia Davis, we expect Cloud-native applications:

- To achieve high-availability while running in fault-prone environments;
- To allow us to continuously release new versions with zero downtime;
- To handle dynamic workloads (e.g. request volumes).

These requirements have a deep impact on the viable solution space for the architecture of our software.

[1] Like the excellent *Cloud-native patterns* by Cornelia Davis!

High availability implies that our application should be able to serve requests with no downtime even if one or more of our machines suddenly starts failing (a *common* occurrence in a Cloud environment[2]). This forces our application to be *distributed* - there should be multiple instances of it running on multiple machines.
The same is true if we want to be able to handle dynamic workloads - we should be able to **measure** if our system is under load and throw more compute at the problem by spinning up new instances of the application. This also requires our infrastructure to be elastic to avoid overprovisioning and its associated costs.
Running a replicated application influences our approach to data persistence - we will avoid using the local filesystem as our primary storage solution, relying instead on databases for our persistence needs.

Zero To Production will thus extensively cover topics that might seem tangential to pure backend application development. But Cloud-native software is all about rainbows and DevOps, therefore we will be spending plenty of time on topics traditionally associated with the craft of **operating** systems.

We will cover how to **instrument** your Rust application to collect logs, traces and metrics to be able to **observe** our system.

We will cover how to set up and evolve your database schema via migrations.

We will cover all the material required to use Rust to tackle both day one and day two concerns of a Cloud-native API.

Working in a team

The impact of those three requirements goes beyond the technical characteristics of our system: it influences how we **build** our software.

To be able to quickly release a new version of our application to our users we need to be sure that our application works.
If you are working on a solo project you can rely on your thorough understanding of the whole system: you wrote it, it might be small enough to fit entirely in your head at any point in time.[3]
If you are working in a team on a commercial project, you will be very often working on code that was neither written or reviewed by you. The original authors might not be around anymore.

You will end up being paralysed by fear every time you are about to introduce changes if you are relying on your comprehensive understanding of what the code does to prevent it from breaking.

You want automated tests.
Running on every commit. On every branch. Keeping `main` healthy.

You want to leverage the type system to make undesirable states difficult or impossible to represent.

[2] For example, many companies run their software on AWS Spot Instances to reduce their infrastructure bills. The price of Spot instances is the result of a continuous auction and it can be substantially cheaper than the corresponding full price for On Demand instances (up to 90% cheaper!).
There is one gotcha: AWS can decommission your Spot instances at any point in time. Your software **must** be fault-tolerant to leverage this opportunity.

[3] Assuming you wrote it recently.
Your past self from one year ago counts as a stranger for all intents and purposes in the world of software development. Pray that your past self wrote comments for your present self if you are about to pick up again an old project of yours.

You want to use every tool at your disposal to empower each member of the team to evolve that piece of software. To contribute fully to the development process even if they might not be as experienced as you or equally familiar with the codebase or the technologies you are using.

Zero To Production will therefore put a strong emphasis on test-driven development and continuous integration from the get-go - we will have a CI pipeline set up before we even have a web server up and running!

We will be covering techniques such as black-box testing for APIs and HTTP mocking - not wildly popular or well documented in the Rust community yet extremely powerful.

We will also borrow terminology and techniques from the Domain Driven Design world, combining them with type-driven design to ensure the correctness of our systems.

> Our main focus is *enterprise software*: correct code which is expressive enough to model the domain and supple enough to support its evolution over time.

We will thus have a bias for boring and correct solutions, even if they incur a performance overhead that could be optimised away with a more careful and chiseled approach.
Get it to run first, optimise it later (if needed).

Who Is This Book For

The Rust ecosystem has had a remarkable focus on smashing adoption barriers with amazing material geared towards beginners and newcomers, a relentless effort that goes from documentation to the continuous polishing of the compiler diagnostics.
There is value in serving the largest possible audience.
At the same time, trying to **always** speak to **everybody** can have harmful side-effects: material that would be relevant to intermediate and advanced users but definitely too much too soon for beginners ends up being neglected.

I struggled with it first-hand when I started to play around with `async/await`.
There was a significant gap between the knowledge I needed to be productive and the knowledge I had built reading *The Rust Book* or working in the Rust numerical ecosystem.
I wanted to get an answer to a straight-forward question:

> Can Rust be a *productive* language for API development?

Yes.

But it can take some time to figure out *how*.
That's why I am writing this book.

I am writing this book for the seasoned backend developers who have read The Rust Book and are now trying to port over a couple of simple systems.

I am writing this book for the new engineers on my team, a trail to help them make sense of the codebases they will contribute to over the coming weeks and months.

I am writing this book for a niche whose needs I believe are currently underserved by the articles and resources available in the Rust ecosystem.

I am writing this book for myself a year ago.
To socialise the knowledge gained during the journey: what does your toolbox look like if you are using Rust for backend development in 2022? What are the design patterns? Where are the pitfalls?

If you do not fit this description but you are working towards it I will do my best to help you on the journey: while we won't be covering a lot of material directly (e.g. most Rust language features) I will try to provide references and links where needed to help you pick up/brush off those concepts along the way.

Let's get started.

Chapter 1

Getting Started

There is more to a programming language than the language itself: tooling is a key element of the *experience* of using the language.
The same applies to many other technologies (e.g. RPC frameworks like gRPC or Apache Avro) and it often has a disproportionate impact on the uptake (or the demise) of the technology itself.

Tooling should therefore be treated as a first-class concern both when designing and teaching the language itself.

The Rust community has put tooling at the forefront since its early days: it shows.
We are now going to take a brief tour of a set of tools and utilities that are going to be useful in our journey. Some of them are officially supported by the Rust organisation, others are built and maintained by the community.

1.1 Installing The Rust Toolchain

There are various ways to install Rust on your system, but we are going to focus on the recommended path: via `rustup`.
Instructions on how to install `rustup` itself can be found at https://rustup.rs.
`rustup` is more than a Rust installer - its main value proposition is *toolchain management*.
A toolchain is the combination of a *compilation target* and a *release channel*.

1.1.1 Compilation Targets

The main purpose of the Rust compiler is to convert Rust code into machine code - a set of instructions that your CPU and operating system can understand and execute.
Therefore you need a different backend of the Rust compiler for each *compilation target*, i.e. for each platform (e.g. 64-bit Linux or 64-bit OSX) you want to produce a running executable for.
The Rust project strives to support a broad range of compilation targets with various level of guarantees. Targets are split into *tiers*, from “guaranteed-to-work” Tier 1 to “best-effort” Tier 3.

An exhaustive and up-to-date list can be found on the Rust forge website.

1.1.2 Release Channels

The Rust compiler itself is a living piece of software: it continuously evolves and improves with the daily contributions of hundreds of volunteers.

The Rust project strives for *stability without stagnation*. Quoting from Rust's documentation:

> *[..]* you should never have to fear upgrading to a new version of stable Rust. Each upgrade should be painless, but should also bring you new features, fewer bugs, and faster compile times.

That is why, for application development, you should generally rely on the latest released version of the compiler to run, build and test your software - the so-called `stable` channel.
A new version of the compiler is released on the `stable` channel every six weeks[1] - the latest version at the time of writing is `v1.80.1`[2].

There are two other release channels:

- `beta`, the candidate for the next release;
- `nightly`, built from the `master` branch of rust-lang/rust every night, thus the name.

Testing your software using the `beta` compiler is one of the many ways to support the Rust project - it helps catching bugs before the release date[3].

`nightly` serves a different purpose: it gives early adopters access to unfinished features[4] before they are released (or even on track to be stabilised!).
I would invite you to think twice if you are planning to run production software on top of the `nightly` compiler: it's called unstable for a reason.

1.1.3 What Toolchains Do We Need?

Installing `rustup` will give you out of the box the latest `stable` compiler with your host platform as a target. `stable` is the release channel that we will be using throughout the book to build, test and run our code.

You can update your toolchains with `rustup update`, while `rustup toolchain list` will give you an overview of what is installed on your system.

We will not need (or perform) any cross-compiling - our production workloads will be running in containers, hence we do not need to cross-compile from our development machine to the target host used in our production environment.

[1] More details on the release schedule can be found in the Rust book.
[2] You can check the next version and its release date at Rust forge.
[3] It's fairly rare for `beta` releases to contain issues thanks to the CI/CD setup of the Rust project. One of its most interesting components is `crater`, a tool designed to scrape crates.io and GitHub for Rust projects to build them and run their test suites to identify potential regressions. Pietro Albini gave an awesome overview of the Rust release process in his Shipping a compiler every six weeks talk at RustFest 2019.
[4] You can check the list of feature flags available on `nightly` in The Unstable Book. *Spoiler*: there are **loads**.

1.2 Project Setup

A toolchain installation via `rustup` bundles together various components.
One of them is the Rust compiler itself, `rustc`. You can check it out with

```
rustc --version
```

You will not be spending a lot of quality time working directly with `rustc` - your main interface for building and testing Rust applications will be `cargo`, Rust's build tool.
You can double-check everything is up and running with

```
cargo --version
```

Let's use `cargo` to create the skeleton of the project we will be working on for the whole book:

```
cargo new zero2prod
```

You should have a new `zero2prod` folder, with the following file structure:

```
zero2prod/
  Cargo.toml
  .gitignore
  .git
  src/
    main.rs
```

The project is already a `git` repository, out of the box.
If you are planning on hosting the project on GitHub, you just need to create a new empty repository and run

```
cd zero2prod
git add .
git commit -am "Project skeleton"
git remote add origin git@github.com:YourGitHubNickName/zero2prod.git
git push -u origin main
```

We will be using GitHub as a reference given its popularity and the recently released GitHub Actions feature for CI pipelines, but you are of course free to choose any other `git` hosting solution (or none at all).

1.3 IDEs

The project skeleton is ready, it is now time to fire up your favourite editor so that we can start messing around with it.
Different people have different preferences but I would argue that the bare minimum you want to have, especially if you are starting out with a new programming language, is a setup that supports syntax highlighting, code navigation and code completion.

Syntax highlighting gives you immediate feedback on glaring syntax errors, while code navigation and code completion enable "exploratory" programming: jumping in and out of the source of your dependencies, quick access to the available methods on a struct or an enum you imported from a crate without having to continuously switch between your editor and docs.rs.

You have two main options for your IDE setup: `rust-analyzer` and RustRover.

1.3.1 Rust-analyzer

`rust-analyzer`[5] is an implementation of the Language Server Protocol for Rust.
The Language Server Protocol makes it easy to leverage `rust-analyzer` in many different editors, including but not limited to VS Code, Emacs, Vim/NeoVim, Zed and Sublime Text 3.

Editor-specific setup instructions can be found on `rust-analyzer`'s website.

1.3.2 RustRover

RustRover provides Rust support to the suite of editors developed by JetBrains[6].
It is free for non-commercial usage.

1.3.3 What Should I Use?

As of September 2024, I recommend using RustRover.
`rust-analyzer` is promising, but it still falls short from offering an IDE experience on par with what RustRover offers today[7].

On the other hand, RustRover forces you to work with a JetBrains' IDE, which you might or might not be willing to. If you'd like to stick to your editor of choice look for its `rust-analyzer` integration/plugin.

It is worth mentioning that `rust-analyzer` is part of a larger library-ification effort taking place within the Rust compiler: there is overlap between `rust-analyzer` and `rustc`, with a lot of duplicated effort.
Evolving the compiler's codebase into a set of re-usable modules will allow `rust-analyzer` to leverage an

[5] `rust-analyzer` is not the first attempt to implement the LSP for Rust: `RLS` was its predecessor. `RLS` took a batch-processing approach: every little change to any of the files in a project would trigger re-compilation of the whole project. This strategy was fundamentally limited and it led to poor performance and responsiveness. RFC2912 formalised the "retirement" of `RLS` as the blessed LSP implementation for Rust in favour of `rust-analyzer`.

[6] RustRover superseded the IntelliJ Rust plugin, which was the first (free) Rust plugin for JetBrains' IDEs.

[7] Neither myself nor the book are sponsored by JetBrains (or any other company). All recommendations, this included, are based on my personal experience.

increasingly larger subset of the compiler codebase, unlocking the on-demand analysis capabilities required to offer a top-notch IDE experience.
An interesting space to keep an eye on in the future[8].

1.4 Inner Development Loop

While working on our project, we will be going through the same steps over and over again:

- Make a change;
- Compile the application;
- Run tests;
- Run the application.

This is also known as the **inner development loop**.
The speed of your inner development loop is as an upper bound on the number of iterations that you can complete in a unit of time.
If it takes 5 minutes to compile and run the application, you can complete at most 12 iterations in an hour. Cut it down to 2 minutes and you can now fit in 30 iterations in the same hour!

Rust does not help us here - compilation speed can become a pain point on big projects. Let's see what we can do to mitigate the issue before moving forward.

1.4.1 Faster Linking

When looking at the inner development loop, we are primarily looking at the performance of incremental compilation - how long it takes `cargo` to rebuild our binary after having made a small change to the source code.
A sizeable chunk of time is spent in the **linking** phase - assembling the actual binary given the outputs of the earlier compilation stages.

The default linker does a good job, but there is a faster alternative: `lld`, a linker developed by the LLVM project.
To speed up the linking phase you have to install the alternative linker on your machine and add this configuration file to the project:

```
# .cargo/config.toml

# On Windows
# ```
# cargo install -f cargo-binutils
# rustup component add llvm-tools-preview
# ```
```

[8]Check their Next Few Years blog post for more details on `rust-analyzer`'s roadmap and main concerns going forward.

```toml
[target.x86_64-pc-windows-msvc]
rustflags = ["-C", "link-arg=-fuse-ld=lld"]

[target.x86_64-pc-windows-gnu]
rustflags = ["-C", "link-arg=-fuse-ld=lld"]

# On Linux:
# - Ubuntu, `sudo apt-get install lld clang`
# - Arch, `sudo pacman -S lld clang`
[target.x86_64-unknown-linux-gnu]
rustflags = ["-C", "linker=clang", "-C", "link-arg=-fuse-ld=lld"]

# On MacOS, `brew install llvm` and follow steps in `brew info llvm`
[target.x86_64-apple-darwin]
rustflags = ["-C", "link-arg=-fuse-ld=lld"]

[target.aarch64-apple-darwin]
rustflags = ["-C", "link-arg=-fuse-ld=/opt/homebrew/opt/llvm/bin/ld64.lld"]
```

There is ongoing work on the Rust compiler to use `lld` as the default linker where possible - soon enough this custom configuration will not be necessary to achieve higher compilation performance![9]

1.4.2 `cargo-watch`

We can also mitigate the impact on our productivity by reducing the **perceived** compilation time - i.e. the time you spend looking at your terminal waiting for `cargo check` or `cargo run` to complete.
Tooling can help here - let's install `cargo-watch`:

```
cargo install cargo-watch
```

`cargo-watch` monitors your source code to trigger commands every time a file changes.
For example:

```
cargo watch -x check
```

will run `cargo check` after every code change.

This reduces the perceived compilation time:

- you are still in your IDE, re-reading the code change you just made;

[9] You can also try the `mold` linker on Linux, and its cousin `sold` on MacOS. They offer very competitive performance, although they are both younger projects and might not be as stable as the alternatives we already mentioned.

- `cargo-watch`, in the meantime, has already kick-started the compilation process;
- once you switch to your terminal, the compiler is already halfway through!

`cargo-watch` supports command chaining as well:

```
cargo watch -x check -x test -x run
```

It will start by running `cargo check`.
If it succeeds, it launches `cargo test`.
If tests pass, it launches the application with `cargo run`.

Our inner development loop, right there!

1.5 Continuous Integration

Toolchain, installed.
Project skeleton, done.
IDE, ready.

One last thing to look at before we get into the details of what we will be building: our **Continuous Integration (CI) pipeline**.

In trunk-based development we should be able to deploy our `main` branch at any point in time.
Every member of the team can branch off from `main`, develop a small feature or fix a bug, merge back into `main` and release to our users.

> Continuous Integration empowers each member of the team to integrate their changes into the main branch multiple times a day.

This has powerful ripple effects.
Some are tangible and easy to spot: it reduces the chances of having to sort out messy merge conflicts due to long-lived branches. Nobody likes merge conflicts.
Some are subtler: **Continuous Integration tightens the feedback loop**. You are less likely to go off on your own and develop for days or weeks just to find out that the approach you have chosen is not endorsed by the rest of the team or it would not integrate well with the rest of the project.
It forces you to engage with your teammates earlier than when it feels comfortable, course-correcting if necessary when it is still easy to do so (and nobody is likely to get offended).

How do we make it possible?

With a collection of automated checks running on every commit - our **CI pipeline**.
If one of the checks fails you cannot merge to `main` - as simple as that.

CI pipelines often go beyond ensuring code health: they are a good place to perform a series of additional important checks - e.g. scanning our dependency tree for known vulnerabilities, linting, formatting, etc.

We will run through the different checks that you might want to run as part of the CI pipeline of your Rust projects, introducing the associated tools as we go along.
We will then provide a set of ready-made CI pipelines for some of the major CI providers.

1.5.1 CI Steps

1.5.1.1 Tests

If your CI pipeline had a single step, it should be testing.
Tests are a first-class concept in the Rust ecosystem and you can leverage `cargo` to run your unit and integration tests:

```
cargo test
```

`cargo test` also takes care of building the project before running tests, hence you do not need to run `cargo build` beforehand (even though most pipelines will invoke `cargo build` before running tests to cache dependencies).

1.5.1.2 Code Coverage

Many articles have been written on the pros and cons of measuring code coverage.
While using code coverage as a quality check has several drawbacks I do argue that it is a quick way to collect information and spot if some portions of the codebase have been overlooked over time and are indeed poorly tested.

The easiest way to measure code coverage of a Rust project is via `cargo-llvm-cov`, a `cargo` subcommand developed by Taiki Endo. You can install `cargo-llvm-cov` with

```
# Additional components required to compute LLVM code coverage
rustup component add llvm-tools-preview
cargo install cargo-llvm-cov
```

while

```
cargo llvm-cov
```

will compute code coverage for your application code.

`cargo-llvm-cov` can be used to upload code coverage metrics to popular services like Codecov or Coveralls - instructions can be found via `cargo llvm-cov --help`.

1.5.1.3 Linting

Writing idiomatic code in any programming language requires time and practice.
It is easy at the beginning of your learning journey to end up with fairly convoluted solutions to problems that could otherwise be tackled with a much simpler approach.

Static analysis can help: in the same way a compiler steps through your code to ensure it conforms to the language rules and constraints, a **linter** will try to spot unidiomatic code, overly-complex constructs and common mistakes/inefficiencies.

The Rust team maintains `clippy`, the official Rust linter[10].
`clippy` is included in the set of components installed by `rustup` if you are using the `default` profile. Often CI environments use `rustup`'s `minimal` profile, which does not include `clippy`.
You can easily install it with

```
rustup component add clippy
```

If it is already installed the command is a no-op.

You can run `clippy` on your project with

```
cargo clippy
```

In our CI pipeline we would like to fail the linter check if `clippy` emits any warnings.
We can achieve it with

```
cargo clippy -- -D warnings
```

Static analysis is not infallible: from time to time `clippy` might suggest changes that you do not believe to be either correct or desirable.
You can mute a warning using the `#[allow(clippy::lint_name)]` attribute on the affected code block or disable the noisy lint altogether for the whole project with a configuration line in `clippy.toml` or a project-level `#![allow(clippy::lint_name)]` directive.
Details on the available lints and how to tune them for your specific purposes can be found in `clippy`'s README.

1.5.1.4 Formatting

Most organizations have more than one line of defence for the `main` branch: one is provided by the CI pipeline checks, the other is often a pull request review.

A lot can be said on what distinguishes a value-adding PR review process from a soul-sucking one - no need to re-open the whole debate here.

[10] Yes, `clippy` is named after the (in)famous paperclip-shaped Microsoft Word assistant.

I know for sure what should **not** be the focus of a good PR review: formatting nitpicks - e.g. *Can you add a newline here?*, *I think we have a trailing whitespace there!*, etc.

Let machines deal with formatting while reviewers focus on architecture, testing thoroughness, reliability, observability. Automated formatting removes a distraction from the complex equation of the PR review process. You might dislike this or that formatting choice, but the complete erasure of formatting bikeshedding is worth the minor discomfort.

rustfmt is the official Rust formatter.
Just like `clippy`, `rustfmt` is included in the set of default components installed by `rustup`. If missing, you can easily install it with

```
rustup component add rustfmt
```

You can format your whole project with

```
cargo fmt
```

In our CI pipeline we will add a formatting step

```
cargo fmt -- --check
```

It will fail when a commit contains unformatted code, printing the difference to the console.[11]

You can tune `rustfmt` for a project with a configuration file, `rustfmt.toml`. Details can be found in `rustfmt`'s README.

1.5.1.5 Security Vulnerabilities

`cargo` makes it very easy to leverage existing crates in the ecosystem to solve the problem at hand.
On the flip side, each of those crates might hide an exploitable vulnerability that could compromise the security posture of your software.

The Rust Secure Code working group maintains an Advisory Database - an up-to-date collection of reported vulnerabilities for crates published on crates.io.

They also provide `cargo-audit`[12], a convenient `cargo` sub-command to check if vulnerabilities have been reported for any of the crates in the dependency tree of your project.
You can install it with

[11]It can be annoying to get a fail in CI for a formatting issue. Most IDEs support a "format on save" feature to make the process smoother. Alternatively, you can use a `git` pre-push hook.

[12]`cargo-deny`, developed by Embark Studios, is another `cargo` sub-command that supports vulnerability scanning of your dependency tree. It also bundles additional checks you might want to perform on your dependencies - it helps you identify unmaintained crates, define rules to restrict the set of allowed software licenses and spot when you have multiple versions of the same crate in your lock file (wasted compilation cycles!). It requires a bit of upfront effort in configuration, but it can be a powerful addition to your CI toolbox.

```
cargo install cargo-audit
```

Once installed, run

```
cargo audit
```

to scan your dependency tree.

We will be running `cargo-audit` as part of our CI pipeline, on every commit.
We will also run it on a daily schedule to stay on top of new vulnerabilities for dependencies of projects that we might not be actively working on at the moment but are still running in our production environment!

1.5.2 Ready-to-go CI Pipelines

> Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime.

Hopefully I have taught you enough to go out there and stitch together a solid CI pipeline for your Rust projects.
We should also be honest and admit that it can take multiple hours of fidgeting around to learn how to use the specific flavour of configuration language used by a CI provider and the debugging experience can often be quite painful, with long feedback cycles.
To speed up the process, you can use these ready-made configuration files for running the steps we just described via GitHub Actions. It is often much easier to tweak an existing setup to suit your specific needs than to write a new one from scratch.

Chapter 2

Building An Email Newsletter

2.1 Our Driving Example

The Foreword stated that

> *Zero To Production* will focus on the challenges of writing cloud-native applications in a team of four or five engineers with different levels of experience and proficiency.

How? Well, *by actually building one!*

2.1.1 Problem-based Learning

Choose a problem you want to solve.
Let the problem drive the introduction of new concepts and techniques.

It flips the hierarchy you are used to: the material you are studying is not relevant because somebody claims it is, it is relevant because it is **useful** to get closer to a solution.
You learn new techniques **and** when it makes sense to reach for them.

The devil is in the details: a problem-based learning path can be delightful, yet it is painfully easy to misjudge how challenging each step of the journey is going to be.
Our driving example needs to be:

- small enough for us to tackle in a book without cutting corners;
- complex enough to surface most of the key themes that come up in bigger systems;
- interesting enough to keep readers engaged as they progress.

We will go for an **email newsletter** - the next section will detail the functionality we plan to cover[1].

[1]Who knows, I might end up using our home-grown newsletter application to release the final chapter - it would definitely provide me with a sense of closure.

2.2 What Should Our Newsletter Do?

There are dozens of companies providing services that include or are centered around the idea of managing a list of email addresses.

While they all share a set of core functionalities (i.e. sending emails), their services are tailored to specific use-cases: UI, marketing spin and pricing will differ significantly between a product targeted at big companies managing hundreds of thousands of addresses with strict security and compliance requirements compared to a SaaS offering geared to indie content creators running their own blogs or small online stores.

Now, we have no ambition to build the next MailChimp or ConvertKit - the scope would definitely be too broad for us to cover over the course of a book. Furthermore, several features would require applying the same concepts and techniques over and over again - it gets tedious to read after a while.

We will try to build an email newsletter service that supports what you need to get off the ground if you are willing to add an email subscription page to your blog[2].

2.2.1 Capturing Requirements: User Stories

The product brief above leaves some room for interpretation - to better scope what our service should support we will leverage *user stories*.
The format is fairly simple:

> As a ...,
> I want to ...,
> So that ...

A user story helps us to capture who we are building for (*as a*), the actions they want to perform (*want to*) as well as their motives (*so that*).

We will fulfill two user stories:

- As a blog visitor,
  I want to subscribe to the newsletter,
  So that I can receive email updates when new content is published on the blog;
- As the blog author,
  I want to send an email to all my subscribers,
  So that I can notify them when new content is published.

We will not add features to

- unsubscribe;

[2] Make no mistake: when buying a SaaS product it is often not the software itself that you are paying for - you are paying for the peace of mind of knowing that there is an engineering team working full time to keep the service up and running, for their legal and compliance expertise, for their security team. We (developers) often underestimate how much time (and headaches) that saves us over time.

- manage multiple newsletters;
- segment subscribers in multiple audiences;
- track opening and click rates.

As said, pretty barebone. We would definitely not be able to launch publicly without giving users the possibility to unsubscribe.
Nonetheless, fulfilling those two stories will give us plenty of opportunities to practice and hone our skills!

2.3 Working In Iterations

Let's zoom on one of those user stories:

> As the blog author,
> I want to send an email to all my subscribers,
> So that I can notify them when new content is published.

What does this mean *in practice*? What do we need to build?
As soon as you start looking closer at the problem tons of questions pop up - e.g. how do we ensure that the caller is indeed the blog author? Do we need to introduce an authentication mechanism? Do we support HTML in emails or do we stick to plain text? What about emojis?

We could easily spend months implementing an extremely polished email delivery system without having even a basic subscribe/unsubscribe functionality in place.
We might become the best at sending emails, but nobody is going to use our email newsletter service - it does not cover the full journey.

Instead of going deep on one story, we will try to build enough functionality to satisfy, *to an extent*, the requirements of all of our stories in our first release.
We will then go back and improve: add fault-tolerance and retries for email delivery, add a confirmation email for new subscribers, etc.

We will work in iterations: each iteration takes a fixed amount of time and gives us a slightly better version of the product, improving the experience of our users.

Worth stressing that we are iterating on product features, not engineering quality: the code produced in each iteration will be tested and properly documented even if it only delivers a tiny, fully functional feature.
Our code is going to production at the end of each iteration - it needs to be production-quality.

2.3.1 Coming Up

Strategy is clear, we can finally get started: the next chapter will focus on the subscription functionality.
Getting off the ground will require some initial heavy-lifting: choosing a web framework, setting up the infrastructure for managing database migrations, putting together our application scaffolding as well as our setup for integration testing.
Expect to spend way more time pair programming with the compiler going forward!

2.4 Checking Your Progress

One last (but crucial!) detail: there is a public GitHub repository for this book.
The GitHub repository hosts all the code for our newsletter API project. It also includes intermediate snapshots, showing what the project looks like at end of each chapter and key sections.
If you get stuck, make sure to compare your code with the one in the repository!

Chapter 3

Sign Up A New Subscriber

We spent the whole previous chapter defining what we will be building (an email newsletter!), narrowing down a precise set of requirements. It is now time to roll up our sleeves and get started with it.

This chapter will take a first stab at implementing this user story:

> As a blog visitor,
> I want to subscribe to the newsletter,
> So that I can receive email updates when new content is published on the blog.

We expect our blog visitors to input their email address in a form embedded on a web page.
The form will trigger an API call to a backend server that will actually process the information, store it and send back a response.
This chapter will focus on that backend server - we will implement the `/subscriptions` POST endpoint.

3.1 Our Strategy

We are starting a new project from scratch - there is a fair amount of upfront heavy-lifting we need to take care of:

- choose a web framework and get familiar with it;
- define our testing strategy;
- choose a crate to interact with our database (we will have to save those emails somewhere!);
- define how we want to manage changes to our database schemas over time (a.k.a. migrations);
- actually write some queries.

That is a lot and jumping in head-first might be overwhelming.
We will add a stepping stone to make the journey more approachable: before tackling `/subscriptions` we will implement a `/health_check` endpoint. No business logic, but a good opportunity to become friends with our web framework and get an understanding of all its different moving parts.

We will be relying on our Continuous Integration pipeline to keep us in check throughout the process - if you have not set it up yet, go back to Chapter 1 and grab one of the ready-made templates.

3.2 Choosing A Web Framework

What web framework should we use to write our Rust API?
You can find many competing options in the ecosystem (`actix-web`, `axum`, `poem`, `tide`, `rocket`, etc.). For this book, we will use `actix-web`.
`actix-web` is one of Rust's oldest frameworks. It has seen extensive production usage, and it has built a large community and plugin ecosystem; last but not least, it runs on `tokio`, therefore minimising the likelihood of having to deal with incompatibilities and interop between different async runtimes.
`actix-web` will therefore be our choice for Zero To Production.

Throughout this chapter and beyond I suggest you to keep a couple of extra browser tabs open: `actix-web`'s website, `actix-web`'s documentation and `actix-web`'s examples collection.

3.3 Our First Endpoint: A Basic Health Check

Let's try to get off the ground by implementing a health-check endpoint: when we receive a `GET` request for `/health_check` we want to return a `200 OK` response with no body.

We can use `/health_check` to verify that the application is up and ready to accept incoming requests.
Combine it with a SaaS service like pingdom.com and you can be alerted when your API goes dark - quite a good baseline for an email newsletter that you are running on the side.

A health-check endpoint can also be handy if you are using a container orchestrator to juggle your application (e.g. Kubernetes or Nomad): the orchestrator can call `/health_check` to detect if the API has become unresponsive and trigger a restart.

3.3.1 Wiring Up `actix-web`

Our starting point will be an *Hello World!* application built with `actix-web`:

```rust
use actix_web::{web, App, HttpRequest, HttpServer, Responder};

async fn greet(req: HttpRequest) -> impl Responder {
    let name = req.match_info().get("name").unwrap_or("World");
    format!("Hello {}!", &name)
}

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    HttpServer::new(|| {
```

```
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
    })
    .bind("127.0.0.1:8000")?
    .run()
    .await
}
```

Let's paste it in our `main.rs` file.
A quick `cargo check`[1]:

```
error[E0432]: unresolved import `actix_web`
 --> src/main.rs:1:5
  |
1 | use actix_web::{web, App, HttpRequest, HttpServer, Responder};
  |     ^^^^^^^^^ use of undeclared type or module `actix_web`

error[E0433]: failed to resolve:
    use of undeclared type or module `tokio`
 --> src/main.rs:8:3
  |
8 | #[tokio::main]
  |   ^^^^^^^^ use of undeclared type or module `tokio`

error: aborting due to 2 previous errors
```

We have not added `actix-web` and `tokio` to our list of dependencies, therefore the compiler cannot resolve what we imported.
We can either fix the situation manually, by adding

```
#! Cargo.toml
# [...]

[dependencies]
actix-web = "4"
tokio = { version = "1", features = ["macros", "rt-multi-thread"] }
```

under `[dependencies]` in our `Cargo.toml` or we can use `cargo add` to quickly add the latest version of both crates as a dependency of our project:

[1] During our development process we are not always interested in producing a runnable binary: we often just want to know if our code compiles or not. `cargo check` was born to serve exactly this usecase: it runs the same checks that are run by `cargo build`, but it does not bother to perform any machine code generation. It is therefore much faster and provides us with a tighter feedback loop. See link for more details.

```
cargo add actix-web@4
cargo add tokio@1 --features macros,rt-multi-thread
```

If you run `cargo check` again there should be no errors.
You can now launch the application with `cargo run` and perform a quick manual test:

```
curl http://127.0.0.1:8000
```

```
Hello World!
```

Cool, it's **alive**!
You can gracefully shut down the web server with `Ctrl+C` if you want to.

3.3.2 Anatomy Of An `actix-web` Application

Let's go back now to have a closer look at what we have just copy-pasted in our `main.rs` file.

```rust
//! src/main.rs
// [...]

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
    })
    .bind("127.0.0.1:8000")?
    .run()
    .await
}
```

3.3.2.1 Server - `HttpServer`

`HttpServer` is the backbone supporting our application. It takes care of things like:

- where should the application be listening for incoming requests? A TCP socket (e.g. `127.0.0.1:8000`)? A Unix domain socket?
- what is the maximum number of concurrent connections that we should allow? How many new connections per unit of time?
- should we enable transport layer security (TLS)?

- etc.

`HttpServer`, in other words, handles all *transport level* concerns.
What happens afterwards? What does `HttpServer` do when it has established a new connection with a client of our API and we need to start handling their requests?
That is where `App` comes into play!

3.3.2.2 Application - `App`

`App` is where all your application logic lives: routing, middlewares, request handlers, etc.
`App` is the component whose job is to take an incoming request as input and spit out a response.
Let's zoom in on our code snippet:

```rust
App::new()
    .route("/", web::get().to(greet))
    .route("/{name}", web::get().to(greet))
```

`App` is a practical example of the *builder pattern*: `new()` gives us a clean slate to which we can add, one bit at a time, new behaviour using a fluent API (i.e. chaining method calls one after the other).
We will cover the majority of `App`'s API surface on a need-to-know basis over the course of the whole book: by the end of our journey you should have touched most of its methods at least once.

3.3.2.3 Endpoint - `Route`

How do we add a new endpoint to our `App`?
The `route` method is probably the simplest way to go about doing it - it is used in a *Hello World!* example after all!

`route` takes two parameters:

- `path`, a string, possibly templated (e.g. `"/{name}"`) to accommodate dynamic path segments;
- `route`, an instance of the `Route` struct.

`Route` combines a *handler* with a set of *guards*.
Guards specify conditions that a request must satisfy in order to "match" and be passed over to the handler. From an implementation standpoint guards are implementors of the `Guard` trait: `Guard::check` is where the magic happens.

In our snippet we have

```rust
.route("/", web::get().to(greet))
```

`"/"` will match all requests without any segment following the base path - i.e. `http://localhost:8000/`.
`web::get()` is a short-cut for `Route::new().guard(guard::Get())` a.k.a. the request should be passed to the handler if and only if its HTTP method is `GET`.

You can start to picture what happens when a new request comes in: `App` iterates over all registered endpoints until it finds a matching one (both path template and guards are satisfied) and passes over the request object to the handler.
This is not 100% accurate but it is a good enough mental model for the time being.

What does a handler look like instead? What is its function signature?
We only have one example at the moment, `greet`:

```
async fn greet(req: HttpRequest) -> impl Responder {
    [...]
}
```

`greet` is an asynchronous function that takes an `HttpRequest` as input and returns *something* that implements the `Responder` trait[2]. A type implements the `Responder` trait if it can be converted into a `HttpResponse` - it is implemented off the shelf for a variety of common types (e.g. strings, status codes, bytes, `HttpResponse`, etc.) and we can roll our own implementations if needed.

Do all our handlers need to have the same function signature of `greet`?
No! `actix-web`, channelling some forbidden trait black magic, allows a wide range of different function signatures for handlers, especially when it comes to input arguments. We will get back to it soon enough.

3.3.2.4 Runtime - `tokio`

We drilled down from the whole `HttpServer` to a `Route`. Let's look again at the whole main function:

```
//! src/main.rs
// [...]

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
    })
    .bind("127.0.0.1:8000")?
    .run()
    .await
}
```

[2] `impl Responder` is using the `impl Trait` syntax introduced in Rust 1.26 - you can find more details in Rust's 2018 edition guide.

What is `#[tokio::main]` doing here? Well, let's remove it and see what happens! `cargo check` screams at us with these errors:

```
error[E0277]: `main` has invalid return type `impl std::future::Future`
 --> src/main.rs:8:20
  |
8 | async fn main() -> Result<(), std::io::Error> {
  |                    ^^^^^^^^^^^^^^^^^^^
  | `main` can only return types that implement `std::process::Termination`
  |
  = help: consider using `()`, or a `Result`

error[E0752]: `main` function is not allowed to be `async`
 --> src/main.rs:8:1
  |
8 | async fn main() -> Result<(), std::io::Error> {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  | `main` function is not allowed to be `async`

error: aborting due to 2 previous errors
```

We need `main` to be asynchronous because `HttpServer::run` is an asynchronous method but `main`, the entrypoint of our binary, **cannot** be an asynchronous function. Why is that?

Asynchronous programming in Rust is built on top of the `Future` trait: a future stands for a value that may not be there *yet*. All futures expose a `poll` method which has to be called to allow the future to make progress and eventually resolve to its final value. You can think of Rust's futures as lazy: unless polled, there is no guarantee that they will execute to completion. This has often been described as a pull model compared to the push model adopted by other languages[3].

Rust's standard library, *by design*, does not include an asynchronous runtime: you are supposed to bring one into your project as a dependency, one more crate under `[dependencies]` in your `Cargo.toml`. This approach is extremely versatile: you are free to implement your own runtime, optimised to cater for the specific requirements of your usecase (see the Fuchsia project or bastion's actor framework).

This explains why `main` cannot be an asynchronous function: who is in charge to call `poll` on it?
There is no special configuration syntax that tells the Rust compiler that one of your dependencies is an asynchronous runtime (e.g. as we do for allocators) and, to be fair, there is not even a standardised definition of what a runtime is (e.g. an `Executor` trait).
You are therefore expected to launch your asynchronous runtime at the top of your `main` function and then use it to drive your futures to completion.
You might have guessed by now what is the purpose of `#[tokio::main]`, but guesses are not enough to satisfy us: we want to *see it*.

[3]Check out the release notes of `async/await` for more details. The talk by withoutboats at Rust LATAM 2019 is another excellent reference on the topic. If you prefer books to talks, check out Futures Explained in 200 Lines of Rust .

How?
`tokio::main` is a procedural macro and this is a great opportunity to introduce `cargo expand`, an awesome addition to our Swiss army knife for Rust development:

Rust macros operate at the token level: they take in a stream of symbols (e.g. in our case, the whole main function) and output a stream of new symbols which then gets passed to the compiler. In other words, the main purpose of Rust macros is **code generation**.
How do we debug or inspect what is happening with a particular macro? You inspect the tokens it outputs!

That is exactly where `cargo expand` shines: it *expands* all macros in your code without passing the output to the compiler, allowing you to step through it and understand what is going on.
Let's use `cargo expand` to demystify `#[tokio::main]`:

```
cargo expand
```

Unfortunately, it fails:

```
error: the option `Z` is only accepted on the nightly compiler
error: could not compile `zero2prod`
```

We are using the `stable` compiler to build, test and run our code. `cargo-expand`, instead, relies on the `nightly` compiler to expand our macros.
You can install the `nightly` compiler by running

```
rustup toolchain install nightly --allow-downgrade
```

Some components of the bundle installed by `rustup` might be broken/missing on the latest `nightly` release: `--allow-downgrade` tells `rustup` to find and install the latest `nightly` where all the needed components are available.

You can use `rustup default` to change the default toolchain used by `cargo` and the other tools managed by `rustup`. In our case, we do not want to switch over to `nightly` - we just need it to be available for `cargo-expand`.

```
# `cargo-expand` will find the `nightly` toolchain automatically
# If we wanted to execute it directly via the `nightly` toolchain,
# we would instead invoke it via `cargo +nightly expand`
cargo expand
```

```
/// [...]

fn main() -> Result<(), std::io::Error> {
    let body = async move {
```

```
        HttpServer::new(|| {
            App::new()
                .route("/", web::get().to(greet))
                .route("/{name}", web::get().to(greet))
        })
        .bind("127.0.0.1:8000")?
        .run()
        .await
    };
    tokio::runtime::Builder::new_multi_thread()
        .enable_all()
        .build()
        .expect("Failed building the Runtime")
        .block_on(body)
}
```

We can finally look at the code after macro expansion!
The `main` function that gets passed to the Rust compiler after `#[tokio::main]` has been expanded is indeed synchronous, which explain why it compiles without any issue.
The key line is this:

```
tokio::runtime::Builder::new_multi_thread()
    .enable_all()
    .build()
    .expect("Failed building the Runtime")
    .block_on(body)
```

We are starting `tokio`'s async runtime and we are using it to drive the future returned by `HttpServer::run` to completion.
In other words, the job of `#[tokio::main]` is to give us the illusion of being able to define an asynchronous `main` while, under the hood, it just takes our `main` asynchronous body and writes the necessary boilerplate to make it run on top of `tokio`'s runtime.

3.3.3 Implementing The Health Check Handler

We have reviewed all the moving pieces in `actix_web`'s *Hello World!* example: `HttpServer`, `App`, `route` and `tokio::main`.
We definitely know enough to modify the example to get our health check working as we expect: return a `200 OK` response with no body when we receive a `GET` request at `/health_check`.

Let's look again at our starting point:

```rust
//! src/main.rs
use actix_web::{web, App, HttpRequest, HttpServer, Responder};

async fn greet(req: HttpRequest) -> impl Responder {
    let name = req.match_info().get("name").unwrap_or("World");
    format!("Hello {}!", &name)
}

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
    })
    .bind("127.0.0.1:8000")?
    .run()
    .await
}
```

First of all we need a request handler. Mimicking `greet` we can start with this signature:

```rust
async fn health_check(req: HttpRequest) -> impl Responder {
    todo!()
}
```

We said that `Responder` is nothing more than a conversion trait into a `HttpResponse`. Returning an instance of `HttpResponse` directly should work then!
Looking at its documentation we can use `HttpResponse::Ok` to get a `HttpResponseBuilder` primed with a 200 status code. `HttpResponseBuilder` exposes a rich fluent API to progressively build out a `HttpResponse` response, but we do not need it here: we can get a `HttpResponse` with an empty body by calling `finish` on the builder.
Gluing everything together:

```rust
async fn health_check(req: HttpRequest) -> impl Responder {
    HttpResponse::Ok().finish()
}
```

A quick `cargo check` confirms that our handler is not doing anything weird. A closer look at `HttpResponseBuilder` unveils that it implements `Responder` as well - we can therefore omit our call to `finish` and shorten our handler to:

```rust
async fn health_check(req: HttpRequest) -> impl Responder {
    HttpResponse::Ok()
}
```

The next step is handler registration - we need to add it to our `App` via `route`:

```rust
App::new()
    .route("/health_check", web::get().to(health_check))
```

Let's look at the full picture:

```rust
//! src/main.rs

use actix_web::{web, App, HttpRequest, HttpResponse, HttpServer, Responder};

async fn health_check(req: HttpRequest) -> impl Responder {
    HttpResponse::Ok()
}

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    HttpServer::new( || {
            App::new()
                .route("/health_check", web::get().to(health_check))
        })
        .bind("127.0.0.1:8000")?
        .run()
        .await
}
```

`cargo check` runs smoothly although it raises one warning:

```
warning: unused variable: `req`
 --> src/main.rs:3:23
  |
3 | async fn health_check(req: HttpRequest) -> impl Responder {
  |                       ^^^
  | help: if this is intentional, prefix it with an underscore: `_req`
  |
  = note: `#[warn(unused_variables)]` on by default
```

Our health check response is indeed static and does not use any of the data bundled with the incoming HTTP request (routing aside). We could follow the compiler's advice and prefix `req` with an underscore... or we could remove that input argument entirely from `health_check`:

```rust
async fn health_check() -> impl Responder {
    HttpResponse::Ok()
}
```

Surprise surprise, it compiles! `actix-web` has some pretty advanced type magic going on behind the scenes and it accepts a broad range of signatures as request handlers - more on that later.

What is left to do?
Well, a little test!

```bash
# Launch the application first in another terminal with `cargo run`
curl -v http://127.0.0.1:8000/health_check
```

```
*   Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8000 (#0)
> GET /health_check HTTP/1.1
> Host: localhost:8000
> User-Agent: curl/7.61.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-length: 0
< date: Wed, 05 Aug 2020 22:11:52 GMT
```

Congrats, you have just implemented your first working `actix_web` endpoint!

3.4 Our First Integration Test

`/health_check` was our first endpoint and we verified everything was working as expected by launching the application and testing it manually via `curl`.

Manual testing though is time-consuming: as our application gets bigger, it gets more and more expensive to manually check that all our assumptions on its behaviour are still valid every time we perform some changes. We'd like to automate as much as possible: those checks should be run in our CI pipeline every time we are committing a change in order to prevent regressions.

While the behaviour of our health check might not evolve much over the course of our journey, it is a good starting point to get our testing scaffolding properly set up.

3.4.1 How Do You Test An Endpoint?

An API is a means to an end: a tool exposed to the outside world to perform some kind of task (e.g. store a document, publish an email, etc.).

The endpoints we expose in our API define the *contract* between us and our clients: a shared agreement about the inputs and the outputs of the system, its *interface*.

The contract might evolve over time and we can roughly picture two scenarios:

- backwards-compatible changes (e.g. adding a new endpoint);
- breaking changes (e.g. removing an endpoint or dropping a field from the schema of its output).

In the first case, existing API clients will keep working as they are. In the second case, existing integrations are likely to break if they relied on the violated portion of the contract.

While we might *intentionally* deploy breaking changes to our API contract, it is critical that we do not break it *accidentally*.

What is the most reliable way to check that we have not introduced a user-visible regression?
Testing the API by interacting with it *in the same exact way* a user would: performing HTTP requests against it and verifying our assumptions on the responses we receive.

This is often referred to as *black box testing*: we verify the behaviour of a system by examining its output given a set of inputs without having access to the details of its internal implementation.

Following this principle, we won't be satisfied by tests that call into handler functions directly - for example:

```rust
#[cfg(test)]
mod tests {
    use crate::health_check;

    #[tokio::test]
    async fn health_check_succeeds() {
        let response = health_check().await;
        // This requires changing the return type of `health_check`
        // from `impl Responder` to `HttpResponse` to compile
        // You also need to import it with `use actix_web::HttpResponse`!
        assert!(response.status().is_success())
    }
}
```

We have not checked that the handler is invoked on GET requests.
We have not checked that the handler is invoked with `/health_check` as the path.

Changing any of these two properties would break our API contract, but our test would still pass - not good enough.

`actix-web` provides some conveniences to interact with an `App` without skipping the routing logic, but there are severe shortcomings to its approach:

- migrating to another web framework would force us to rewrite our whole integration test suite. As much as possible, we'd like our integration tests to be *highly decoupled* from the technology underpinning our API implementation (e.g. having framework-agnostic integration tests is life-saving when you are going through a large rewrite or refactoring!);
- due to some `actix-web`'s limitations[4], we wouldn't be able to share our `App` startup logic between our production code and our testing code, therefore undermining our trust in the guarantees provided by our test suite due to the risk of divergence over time.

We will opt for a fully black-box solution: we will launch our application at the beginning of each test and interact with it using an off-the-shelf HTTP client (e.g. `reqwest`).

3.4.2 Where Should I Put My Tests?

Rust gives you three options when it comes to writing tests:

- next to your code in an *embedded test module*, e.g.

```rust
// Some code I want to test

#[cfg(test)]
mod tests {
    // Import the code I want to test
    use super::*;

    // My tests
}
```

- in an external `tests` folder, i.e.

```
src/
tests/
Cargo.toml
Cargo.lock
 ...
```

- as part of your public documentation (*doc tests*), e.g.

[4] `App` is a generic struct and some of the types used to parametrise it are private to the `actix_web` project. It is therefore impossible (or, at least, so cumbersome that I have never succeeded at it) to write a function that returns an instance of `App`.

```rust
/// Check if a number is even.
/// ```rust
/// use zero2prod::is_even;
///
/// assert!(is_even(2));
/// assert!(!is_even(1));
/// ```
pub fn is_even(x: u64) -> bool {
    x % 2 == 0
}
```

What is the difference?
An embedded test module is part of your project, just hidden behind a configuration conditional check, `#[cfg(test)]`. Anything under the `tests` folder and your documentation tests, instead, are compiled in their own separate binaries.
This has consequences when it comes to *visibility* rules.

An embedded test module has privileged access to the code living next to it: it can interact with structs, methods, fields and functions that have not been marked as public and would normally not be available to a user of our code if they were to import it as a dependency of their own project.
Embedded test modules are quite useful for what I call *iceberg projects*, i.e. the exposed surface is very limited (e.g. a couple of public functions), but the underlying machinery is much larger and fairly complicated (e.g. tens of routines). It might not be straight-forward to exercise all the possible edge cases via the exposed functions - you can then leverage embedded test modules to write unit tests for private sub-components to increase your overall confidence in the correctness of the whole project.

Tests in the external `tests` folder and doc tests, instead, have exactly the same level of access to your code that you would get if you were to add your crate as a dependency in another project. They are therefore used mostly for *integration testing*, i.e. testing your code by calling it in the same exact way a user would.

Our email newsletter is not a library, therefore the line is a bit blurry - we are not exposing it to the world as a Rust crate, we are putting it out there as an API accessible over the network.
Nonetheless we are going to use the `tests` folder for our API integration tests - it is more clearly separated and it is easier to manage test helpers as sub-modules of an external test binary.

3.4.3 Changing Our Project Structure For Easier Testing

We have a bit of housekeeping to do before we can actually write our first test under `/tests`.
As we said, anything under `tests` ends up being compiled in its own binary - all our code under test is imported as a crate. But our project, at the moment, is a *binary*: it is meant to be executed, not to be shared. Therefore we can't import our `main` function in our tests as it is right now.

If you won't take my word for it, we can run a quick experiment:

```bash
# Create the tests folder
mkdir -p tests
```

Create a new `tests/health_check.rs` file with

```rust
//! tests/health_check.rs

use zero2prod::main;

#[test]
fn dummy_test() {
    main()
}
```

`cargo test` should fail with something similar to

```
error[E0432]: unresolved import `zero2prod`
 --> tests/health_check.rs:1:5
  |
1 | use zero2prod::main;
  |     ^^^^^^^^^ use of undeclared type or module `zero2prod`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0432`.
error: could not compile `zero2prod`.
```

We need to refactor our project into a library and a binary: all our logic will live in the library crate while the binary itself will be just an entrypoint with a very slim `main` function.
First step: we need to change our `Cargo.toml`.
It currently looks something like this:

```toml
[package]
name = "zero2prod"
version = "0.1.0"
authors = ["Luca Palmieri <contact@lpalmieri.com>"]
edition = "2021"

[dependencies]
# [...]
```

We are relying on `cargo`'s default behaviour: unless something is spelled out, it will look for a `src/main.rs` file as the binary entrypoint and use the `package.name` field as the binary name.
Looking at the manifest target specification, we need to add a `lib` section to add a library to our project:

```toml
[package]
name = "zero2prod"
version = "0.1.0"
authors = ["Luca Palmieri <contact@lpalmieri.com>"]
edition = "2021"

[lib]
# We could use any path here, but we are following the community convention
# We could specify a library name using the `name` field. If unspecified,
# cargo will default to `package.name`, which is what we want.
path = "src/lib.rs"

[dependencies]
# [ ... ]
```

The `lib.rs` file does not exist yet and `cargo` won't create it for us:

```
cargo check
```

```
error: couldn't read src/lib.rs: No such file or directory (os error 2)

error: aborting due to previous error

error: could not compile `zero2prod`
```

Let's add it then - it can be empty for now.

```bash
touch src/lib.rs
```

Everything should be working now: `cargo check` passes and `cargo run` still launches our application. Although *it is working*, our `Cargo.toml` file now does not give you at a glance the full picture: you see a library, but you don't see our binary there. Even if not strictly necessary, I prefer to have everything spelled out as soon as we move out of the auto-generated vanilla configuration:

```toml
[package]
name = "zero2prod"
version = "0.1.0"
authors = ["Luca Palmieri <contact@lpalmieri.com>"]
edition = "2021"

[lib]
```

```toml
path = "src/lib.rs"

# Notice the double square brackets: it's an array in TOML's syntax.
# We can only have one library in a project, but we can have multiple binaries!
# If you want to manage multiple libraries in the same repository
# have a look at the workspace feature - we'll cover it later on.
[[bin]]
path = "src/main.rs"
name = "zero2prod"

[dependencies]
# [...]
```

Feeling nice and clean, let's move forward.
For the time being we can move our `main` function, as it is, to our library (named `run` to avoid clashes):

```rust
//! main.rs

use zero2prod::run;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    run().await
}
```

```rust
//! src/lib.rs

use actix_web::{web, App, HttpResponse, HttpServer};

async fn health_check() -> HttpResponse {
    HttpResponse::Ok().finish()
}

// We need to mark `run` as public.
// It is no longer a binary entrypoint, therefore we can mark it as async
// without having to use any proc-macro incantation.
pub async fn run() -> Result<(), std::io::Error> {
    HttpServer::new(|| {
            App::new()
                .route("/health_check", web::get().to(health_check))
        })
        .bind("127.0.0.1:8000")?
        .run()
```

```
        .await
}
```

Alright, we are ready to write some juicy integration tests!

3.5 Implementing Our First Integration Test

Our spec for the health check endpoint was:

> When we receive a `GET` request for `/health_check` we return a `200 OK` response with no body.

Let's translate that into a test, filling in as much of it as we can:

```rust
//! tests/health_check.rs

// `tokio::test` is the testing equivalent of `tokio::main`.
// It also spares you from having to specify the `#[test]` attribute.
//
// You can inspect what code gets generated using
// `cargo expand --test health_check` (<- name of the test file)
#[tokio::test]
async fn health_check_works() {
    // Arrange
    spawn_app().await.expect("Failed to spawn our app.");
    // We need to bring in `reqwest`
    // to perform HTTP requests against our application.
    let client = reqwest::Client::new();

    // Act
    let response = client
        .get("http://127.0.0.1:8000/health_check")
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert!(response.status().is_success());
    assert_eq!(Some(0), response.content_length());
}

// Launch our application in the background ~somehow~
async fn spawn_app() -> Result<(), std::io::Error> {
```

```
    todo!()
}
```

```
#! Cargo.toml
# [...]
# Dev dependencies are used exclusively when running tests or examples
# They do not get included in the final application binary!
[dev-dependencies]
reqwest = "0.12"
# [...]
```

Take a second to *really* look at this test case.
`spawn_app` is the only piece that will, reasonably, depend on our application code.
Everything else is *entirely decoupled from the underlying implementation details* - if tomorrow we decide to ditch Rust and rewrite our application in Ruby on Rails we can still use the same test suite to check for regressions in our new stack as long as `spawn_app` gets replaced with the appropriate trigger (e.g. a bash command to launch the Rails app).

The test also covers the full range of properties we are interested to check:

- the health check is exposed at `/health_check`;
- the health check is behind a GET method;
- the health check always returns a 200;
- the health check's response has no body.

If this passes we are done.

The test as it is crashes before doing anything useful: we are missing `spawn_app`, the last piece of the integration testing puzzle.
Why don't we just call `run` in there? I.e.

```
//! tests/health_check.rs
// [...]

async fn spawn_app() -> Result<(), std::io::Error> {
    zero2prod::run().await
}
```

Let's try it out!

```
cargo test
```

```
     Running target/debug/deps/health_check-fc74836458377166

running 1 test
test health_check_works ...
test health_check_works has been running for over 60 seconds
```

No matter how long you wait, test execution will never terminate. What is going on?

In `zero2prod::run` we invoke (and await) `HttpServer::run`. `HttpServer::run` returns an instance of `Server` - when we call `.await` it starts listening on the address we specified *indefinitely*: it will handle incoming requests as they arrive, but it will never shutdown or "complete" on its own.
This implies that `spawn_app` never returns and our test logic never gets executed.

We need to run our application *as a background task*.
`tokio::spawn` comes quite handy here: `tokio::spawn` takes a future and hands it over to the runtime for polling, without waiting for its completion; it therefore runs *concurrently* with downstream futures and tasks (e.g. our test logic).

Let's refactor `zero2prod::run` to return a `Server` without awaiting it:

```rust
//! src/lib.rs

use actix_web::{web, App, HttpResponse, HttpServer};
use actix_web::dev::Server;

async fn health_check() -> HttpResponse {
    HttpResponse::Ok().finish()
}

// Notice the different signature!
// We return `Server` on the happy path and we dropped the `async` keyword
// We have no .await call, so it is not needed anymore.
pub fn run() -> Result<Server, std::io::Error> {
    let server = HttpServer::new(|| {
            App::new()
                .route("/health_check", web::get().to(health_check))
        })
        .bind("127.0.0.1:8000")?
        .run();
    // No .await here!
    Ok(server)
}
```

We need to amend our `main.rs` accordingly:

```rust
//! src/main.rs

use zero2prod::run;

#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    // Bubble up the io::Error if we failed to bind the address
    // Otherwise call .await on our Server
    run()?.await
}
```

A quick `cargo check` should reassure us that everything is in order.
We can now write `spawn_app`:

```rust
//! tests/health_check.rs
// [...]

// No .await call, therefore no need for `spawn_app` to be async now.
// We are also running tests, so it is not worth it to propagate errors:
// if we fail to perform the required setup we can just panic and crash
// all the things.
fn spawn_app() {
    let server = zero2prod::run().expect("Failed to bind address");
    // Launch the server as a background task
    // tokio::spawn returns a handle to the spawned future,
    // but we have no use for it here, hence the non-binding let
    let _ = tokio::spawn(server);
}
```

Quick adjustment to our test to accommodate the changes in `spawn_app`'s signature:

```rust
//! tests/health_check.rs
// [...]

#[tokio::test]
async fn health_check_works() {
    // No .await, no .expect
    spawn_app();
    // [...]
}
```

It's time, let's run that `cargo test` command!

```
cargo test
```

```
     Running target/debug/deps/health_check-a1d027e9ac92cd64

running 1 test
test health_check_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Yay! Our first integration test is green!
Give yourself a pat on the back on my behalf for the second major milestone in the span of a single chapter.

3.5.1 Polishing

We got it working, now we need to have a second look and improve it, if needed or possible.

3.5.1.1 Clean Up

What happens to our app running in the background when the test run ends? Does it shut down? Does it linger as a zombie somewhere?

Well, running `cargo test` multiple times in a row always succeeds - a strong hint that our 8000 port is getting released at the end of each run, therefore implying that the application is correctly shut down.
A second look at `tokio::spawn`'s documentation supports our hypothesis: when a `tokio` runtime is shut down all tasks spawned on it are dropped. `tokio::test` spins up a new runtime at the beginning of each test case and they shut down at the end of each test case.
In other words, good news - no need to implement any clean up logic to avoid leaking resources between test runs.

3.5.1.2 Choosing A Random Port

`spawn_app` will always try to run our application on port 8000 - not ideal:

- if port 8000 is being used by another program on our machine (e.g. our own application!), tests will fail;
- if we try to run two or more tests in parallel only one of them will manage to bind the port, all others will fail.

We can do better: tests should run their background application on a random available port.
First of all we need to change our `run` function - it should take the application address as an argument instead of relying on a hard-coded value:

```rust
//! src/lib.rs
// [...]

pub fn run(address: &str) -> Result<Server, std::io::Error> {
    let server = HttpServer::new(|| {
            App::new()
                .route("/health_check", web::get().to(health_check))
        })
        .bind(address)?
        .run();
    Ok(server)
}
```

All `zero2prod::run()` invocations must then be changed to `zero2prod::run("127.0.0.1:8000")` to preserve the same behaviour and get the project to compile again.

How do we find a random available port for our tests?
The operating system comes to the rescue: we will be using port 0.
Port 0 is special-cased at the OS level: trying to bind port 0 will trigger an OS scan for an available port which will then be bound to the application.

It is therefore enough to change `spawn_app` to

```rust
//! tests/health_check.rs
// [...]

fn spawn_app() {
    let server = zero2prod::run("127.0.0.1:0").expect("Failed to bind address");
    let _ = tokio::spawn(server);
}
```

Done - the background app now runs on a random port every time we launch `cargo test`!
There is only a small issue... our test is failing[5]!

```
running 1 test
test health_check_works ... FAILED

failures:

---- health_check_works stdout ----
thread 'health_check_works' panicked at
```

[5]There is a remote chance that the OS ended up picking 8000 as random port and everything worked out smoothly. Cheers to you lucky reader!