Luca Palmieri

ZERO TO PRODUCTION IN RUST

AN OPINIONATED INTRODUCTION TO BACKEND DEVELOPMENT

Contents

Foreword

What Is This Book About

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

The socio-technical 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 master.

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.

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.

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

[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.

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 master healthy.

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

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 initial response to a random tweet about the *idea* of spinning up this project has been overwhelming.
Hundreds of subscribers to a rushed email newsletter.
Several emails and private messages detailing this or that specific setup and the challenges they are currently facing.

I have written and re-written the table of contents more than a couple of times trying to zero in on what I want to cover. I realised by the second version of it that I can't satisfy many of those emails and DMs.

[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.

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 2020? 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.

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 here.

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[4] - the latest version at the time of writing is `v1.43.1`[5].

There are two other release channels:

- `beta`, the candidate for the next release;

[4] More details on the release schedule can be found here.
[5] You can check the next version and its release date at Rust forge.

- `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[6].

`nightly` serves a different purpose: it gives early adopters access to unfinished features[7] 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.

Some of the tools we will be using on our development machine (e.g macro expansion) will rely on the `nightly` compiler. While `nightly` is discouraged for production workloads it is not a big deal if something fails on our local machine - we can live with that.

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.

We are only specifying the release channel, `nightly` - `rustup` uses our host platform as default for the target. On my system, if I wanted to be explicit, I would have to use

```
rustup toolchain install \
    nightly-x86_64-unknown-linux-gnu \
    --allow-downgrade
```

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.

1.2 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
```

[6] 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.

[7] You can check the list of feature flags available on `nightly` in The Unstable Book. *Spoiler*: there are **loads**.

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 master
```

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 IntelliJ Rust.

1.3.1 Rust-analyzer

`rust-analyzer`[8] 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 and Sublime Text 3.

Editor-specific setup instructions can be found here.

1.3.2 IntelliJ Rust

IntelliJ Rust provides Rust support to the suite of editors developed by JetBrains.

If you don't have a JetBrains license[9], IntelliJ IDEA is available for free and supports IntelliJ Rust.
If you have a JetBrains license, CLion is your go-to editor for Rust in JetBrains' IDE suite.

[8] `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`.

[9] Students and teachers can claim a free JetBrains educational license.

1.3.3 What Should I Use?

As of May 2020, IntelliJ Rust should be preferred.
Although `rust-analyzer` is promising and has shown incredible progress over the last year, it is still quite far from delivering an IDE experience on par with what IntelliJ Rust offers today.

On the other hand, IntelliJ Rust 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 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[10].

1.4 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 `master` branch at any point in time.
Every member of the team can branch off from `master`, develop a small feature or fix a bug, merge back into `master` 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's 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 `master` - 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.

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

1.4.1 CI Steps

1.4.1.1 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).

1.4.1.2 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 tarpaulin`, a `cargo` subcommand developed by xd009642. You can install `tarpaulin` with

```
# At the time of writing tarpaulin only supports
# x86_64 CPU architectures running Linux.
cargo install cargo-tarpaulin
```

while

```
cargo tarpaulin --ignore-tests
```

will compute code coverage for you application code, ignoring your test functions.

`tarpaulin` can be used to upload code coverage metrics to popular services like Codecov or Coveralls - instructions can be found in `tarpaulin`'s README.

1.4.1.3 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[11].
`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.

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

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.

1.4.1.4 Formatting

Most organizations have more than one line of defence for the `master` 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.
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.

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

1.4.1.5 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

```
cargo install cargo-audit
```

Once installed, run

```
cargo audit
```

to scan your dependency tree.

[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.

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!

1.4.2 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.
I have thus decided to collect a set of ready-made configuration files for the most popular CI providers - the exact steps we just described, ready to be embedded in your project repository:

- GitHub Actions;
- CircleCI;
- GitLab CI;
- Travis.

It is often much easier to tweak an existing setup to suite your specific needs than to write a new one from scratch.
Feel free to get in touch if you would like to provide a template for a CI provider that is currently not covered in the list above.

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[13].

2.1.2 Course-correcting

Problem-based learning works best in an interactive environment: the teacher acts as a facilitator, providing more or less support based on the behavioural cues and reactions of the participants.
A book, published on a website, does not give me the same chance.
I truly appreciate feedback on the material - please reach out to contact@lpalmieri.com or send me a DM on Twitter.
Providing feedback is, at this stage, a tangible way to contribute to *Zero To Production*.

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.

[13] 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.

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 - nothing more, nothing less[14].

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 three 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;
- As a subscriber,
  I want to be able to unsubscribe from the newsletter,
  So that I can stop receiving email updates from the blog.

We will not add features to

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

As said, pretty barebone - nonetheless, enough to satisfy the requirements of most blog authors. It would certainly satisfy mine for *Zero To Production* itself.

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.

[14] 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.

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 code 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!

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, have a quick look at Chapter 1 (or grab one of the ready-made templates).

3.2 Choosing A Web Framework

What web framework should we use to write our Rust API?
This was supposed to be a section on the pros and cons of the Rust web frameworks currently available. It eventually grew to be so long that it did not make sense to embed it here and I published it as a spin-off article: check out *Choosing a Rust web framework, 2020 edition* for a deep-dive on `actix-web`, `rocket`, `tide` and `warp`.

TL;DR: as of August 2020, `actix-web` should be your go-to web framework when it comes to Rust APIs aimed for production usage - it has seen extensive usage in the past couple of years, it has a large and healthy community behind it and it runs on `tokio`, therefore minimising the likelihood of having to deal with incompatibilities/interop between different async runtimes.
It will thus be our choice for Zero To Production.

Nonetheless `tide`, `rocket` and `warp` have huge potential and we might end up making a different decision in 2021 - if you are following along Zero To Production using a different framework I'd be delighted to have a look at your code! Please shoot me an email at *contact@lpalmieri.com*

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 the *Hello World!* example on `actix-web`'s homepage:

```
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)
}

#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    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`[15]:

```
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 `actix_rt`
 --> src/main.rs:8:3
  |
8 | #[actix_rt::main]
  |   ^^^^^^^^ use of undeclared type or module `actix_rt`

error: aborting due to 2 previous errors
```

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

[15] 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.

```
actix-web = "2.0.0"
actix-rt = "1.1.1"
```

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:

```
cargo add actix-web
cargo add actix-rt
```

`cargo add` is not a default `cargo` command: it is provided by `cargo-edit`, a community-maintained[16] `cargo` extension. You can install it with: `bash cargo install cargo-edit` 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://localhost:8000
```

```
Hello World!
```

Cool, it's **alive**!

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.

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    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 level 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:

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

[16] `cargo` follows the same philosophy of Rust's standard library: where possible, the addition of new functionality is explored via third-party crates and then upstreamed where it makes sense to do so (e.g. `cargo-vendor`).

`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 an *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

```
.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[17]. A type implements the `Responder` trait if it can be converted (asynchronously) 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 - `actix_rt`

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

```
#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
```

[17] `impl Responder` is using the `impl Trait` syntax introduced in Rust 1.26 - you can find more details here.

```
    })
    .bind("127.0.0.1:8000")?
    .run()
    .await
}
```

What is `#[actix_rt::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() -> std::io::Result<()> {
  |                    ^^^^^^^^^^^^^^^^^^^ `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() -> std::io::Result<()> {
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ `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 is often been described as a pull model compared to the push model adopted by other languages[18].

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 `#[actix_rt::main]`, but guesses are not enough to satisfy us: we want to *see it*.

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

```
cargo install cargo-expand
```

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**.

[18] 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 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 demistify `#[actix_rt::main]`:

```
cargo expand
```

```
/// [...]

fn main() -> std::io::Result<()> {
    actix_rt::System::new("main").block_on(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
        }
    })
}
```

The `main` function that gets passed to the Rust compiler after `#[actix_rt::main]` has been expanded is indeed synchronous, which explain why it compiles without any issue.
The key line is this:

```
actix_rt::System::new("main").block_on(async move { ... })
```

We are starting `actix`'s async runtime (rt = runtime) and we are using it to drive the future returned by `HttpServer::run` to completion.
In other words, the job of `#[actix_rt::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 `actix`'s runtime.

`actix`'s runtime is built on top of `tokio`'s: we can therefore leverage the whole `tokio` ecosystem when building our applications.

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 `actix_rt::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:

```
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)
}

#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    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:

```
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:

```
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:

```
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`:

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

Let's look at the full picture:

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

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

#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    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`:

```
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!

```
# Launch the application first in another terminal with `cargo run`
curl -v http://localhost: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:

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

    #[actix_rt::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
        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[19], 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.

```
// Some code I want to test

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

[19] `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`.

```
    // My tests
}
```

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

```
> ls

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

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

```
/// 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:

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

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

```
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:

```
[package]
name = "zero2prod"
version = "0.1.0"
authors = ["Luca Palmieri <contact@lpalmieri.com>"]
edition = "2018"

[dependencies]
actix-web = "2.0.0"
actix-rt = "1.1.1"
```

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:

```
[package]
name = "zero2prod"
version = "0.1.0"
authors = ["Luca Palmieri <contact@lpalmieri.com>"]
edition = "2018"

[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]
actix-web = "2.0.0"
actix-rt = "1.1.1"
```

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.

```
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:

```
[package]
name = "zero2prod"
version = "0.1.0"
authors = ["Luca Palmieri <contact@lpalmieri.com>"]
edition = "2018"

[lib]
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 = "app"

[dependencies]
actix-web = "2.0.0"
actix-rt = "1.1.1"
```

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):

```
// main.rs
use zero2prod::run;

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

```
// 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() -> std::io::Result<()> {
    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:

```
// tests/health_check.rs

// `actix_rt::test` is the testing equivalent of `actix_rt::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)
#[actix_rt::test]
async fn health_check_works() {
    // Arrange
    spawn_app().await.expect("Failed to spawn our app.");
    // We brought `reqwest` in as a _development_ dependency
    // to perform HTTP requests against our application.
    // Either add it manually under [dev-dependencies] in Cargo.toml
    // or run `cargo add reqwest --dev`
    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() -> std::io::Result<()> {
    todo!()
}
```

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 out 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.

```
async fn spawn_app() -> std::io::Result<()> {
    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`[20] 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:

```
// 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:

[20] A reminder that `actix_rt`'s runtime is layered on top of `tokio`'s, hence all `tokio` primitives will work like a charm because they are being invoked and executed in the context of a running `tokio` runtime.

```
use zero2prod::run;

#[actix_rt::main]
async fn main() -> std::io::Result<()> {
    // 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`:

```
// 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() {
    // New dev dependency - let's add tokio to the party with
    // `cargo add tokio --dev`
    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:

```
#[actix_rt::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. `actix_rt::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:

```
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

```
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[21]!

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

failures:

---- health_check_works stdout ----
thread 'health_check_works' panicked at
    'Failed to execute request.:
        reqwest::Error { kind: Request, url: "http://localhost:8000/health_check",
        source: hyper::Error(
            Connect,
            ConnectError(
                "tcp connect error",
                Os {
                    code: 111,
                    kind: ConnectionRefused,
                    message: "Connection refused"
                }
            )
```

[21] 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!

```
        )
    }', tests/health_check.rs:10:20
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Panic in Arbiter thread.


failures:
    health_check_works

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

Our HTTP client is still calling `127.0.0.1:8000` and we really don't know what to put there now: the application port is determined at runtime, we cannot hard code it there.
We need, somehow, to find out what port the OS has gifted our application and return it from `spawn_app`.

There are a few ways to go about it - we will use a `std::net::TcpListener`.
Our `HttpServer` right now is doing double duty: given an address, it will bind it and then start the application. We can take over the first step: we will bind the port on our own with `TcpListener` and then hand that over to the `HttpServer` using `listen`.

What is the upside?
`TcpListener::local_addr` returns a `SocketAddr` which exposes the actual port we bound via `.port()`.

Let's begin with our `run` function:

```
use actix_web::dev::Server;
use actix_web::{web, App, HttpResponse, HttpServer};
use std::net::TcpListener;

// [...]

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

The change broke both our `main` and our `spawn_app` function. I'll leave `main` to you, let's focus on `spawn_app`:

```
fn spawn_app() -> String {
    let listener = TcpListener::bind("127.0.0.1:0")
        .expect("Failed to bind random port");
    // We retrieve the port assigned to us by the OS
    let port = listener.local_addr().unwrap().port();
    let server = zero2prod::run(listener).expect("Failed to bind address");
    let _ = tokio::spawn(server);
    // We return the application address to the caller!
    format!("http://127.0.0.1:{}", port)
}
```

We can now leverage the application address in our test to point our `reqwest::Client`:

```
#[actix_rt::test]
async fn health_check_works() {
    // Arrange
    let address = spawn_app();
    let client = reqwest::Client::new();
```

```
    // Act
    let response = client
        // Use the returned application address
        .get(&format!("{}/health_check", &address))
        .send()
        .await
        .expect("Failed to execute request.");

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

All is good - `cargo test` comes out green. Our setup is much more robust now!

3.6 Refocus

Let's take a small break to look back, we covered a fair amount of ground!
We set out to implement a `/health_check` endpoint and that gave us the opportunity to learn more about the fundamentals of our web framework, `actix-web`, as well as the basics of (integration) testing for Rust APIs.

It is now time to capitalise on what we learned to finally fulfill the first user story of our email newsletter project:

> 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 a `POST /subscribtions` call to our backend API that will actually process the information, store it and send back a response.

We will have to dig into:

- how to read data collected in a HTML form in `actix-web` (i.e. how do I parse the request body of a `POST`?);
- what libraries are available to work with a PostgreSQL database in Rust (`diesel` vs `sqlx` vs `tokio-postgres`);
- how to setup and manage migrations for our database;
- how to get our hands on a database connection in our API request handlers;
- how to test for side-effects (a.k.a. stored data) in our integration tests;
- how to avoid weird interactions between tests when working with a database.

Let's get started!

3.7 Working With HTML Forms

3.7.1 Refining Our Requirements

What information should we collect from a visitor in order to enroll them as a subscriber of our email newsletter?

Well, we certainly need their email addresses (it is an *email* newsletter after all).
What else?