Luca Palmieri

ZERO TO PRODUCTION IN RUST

AN OPINIONATED INTRODUCTION TO BACKEND DEVELOPMENT

Sold to
oliverleung2@gmail.com

Contents

5 Going Live 123

6 Reject Invalid Subscribers #1 151

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.

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

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

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.

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

[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.

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 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[1] - the latest version at the time of writing is `v1.59.0`[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 here.

[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 IntelliJ Rust.

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 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[6], 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.

1.3.3 What Should I Use?

As of March 2022, 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[7].

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

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

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

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 are faster alternatives depending on the operating system you are using:

- `lld` on Windows and Linux, a linker developed by the LLVM project;
- `zld` on MacOS.

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
# ```
[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 michaeleisel/zld/zld`
[target.x86_64-apple-darwin]
rustflags = ["-C", "link-arg=-fuse-ld=/usr/local/bin/zld"]

[target.aarch64-apple-darwin]
rustflags = ["-C", "link-arg=-fuse-ld=/usr/local/bin/zld"]
```

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![8]

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;
- `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.

[8] This might not truly be the case though - `mold` is the newest linker on the block and it looks even faster than `lld`! It feels a bit early, so we will not be using it as our default linker, but consider checking it out.

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 tarpaulin`, a `cargo` sub-command 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 your 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.

1.5.1.3 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[9].
`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.
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 equa-

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

tion 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.[10]

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`[11], 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.

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!

[10] 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.

[11] `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.

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.
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 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].

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.

[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.

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.

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[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 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;

[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.

- 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.

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!

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, 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 March 2022, `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 later in 2022 - 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 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]
```

```rust
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`[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

```toml
#! 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:

```
cargo add actix-web@4.0.0
```

`cargo add` is not a default `cargo` command: it is provided by `cargo-edit`, a community-maintained[2] `cargo` extension. You can install it with:

[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.

[2] `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`).

```
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://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() -> 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 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`:

```rust
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[3]. 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.

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

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:

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

#[tokio::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
}
```

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() -> 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 has often been described as a pull model compared to the push model adopted by other languages[4].

[4] Check out the release notes of `async/await` for more details. The talk by withoutboats at Rust LATAM 2019

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*.

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:

```
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**.
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 for

is another excellent reference on the topic. If you prefer books to talks, check out Futures Explained in 200 Lines of Rust.

cargo-expand.
Luckily enough, cargo allows us to specify the toolchain on a per-command basis:

```
# Use the nightly toolchain just for this command invocation
cargo +nightly expand
```

```
/// [...]

fn main() -> std::io::Result<()> {
    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:

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

```rust
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() -> 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:

```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() -> 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`:

```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[5], 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.

```
> ls

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

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

```rust
/// Check if a number is even.
/// ```rust
/// use zero2prod::is_even;
///
```

[5] `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`.

```
/// 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

```
//! 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:

```
[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:

```
[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.

```
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 = "2021"

[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 = "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):

```
//! main.rs

use zero2prod::run;

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

```rust
//! 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:

```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() -> std::io::Result<()> {
    todo!()
}
```

```toml
#! 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 = "0.11"
# [...]
```

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.

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

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

Let's try it out!

```bash
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() -> 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`:

```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:

```
//! 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[6]!

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

[6] 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!

```
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"
                }
            )
        )
    }', 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:

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

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))
```

```rust
        })
        .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`:

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

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`:

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

#[tokio::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 /subscriptions` 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?

This would usually spark a conversation among the engineers on the team as well as the product manager in your typical business setup. In this case, we are both the technical leads and the product owners so we get to call the shots!

Speaking from personal experience, people generally use throwaway or masked emails when subscribing to newsletters (or, at least, most of you did when subscribing to Zero To Production!).
It would thus be nice to collect a `name` that we could use for our email greetings (the infamous `Hey {{subscriber.name}}!`) as well as to spot mutuals or people we know in the list of subscribers.
We are not cops, we have no interest in the `name` field being *authentic* - we will let people input whatever they feel like using as their identifier in our newsletter system: `DenverCoder9`, we welcome you.

It is settled then: we want an email address and a name for all new subscribers.

Given that the data is collected via a HTML form, it will be passed to our backend API in the body of a `POST` request. How is the body going to be encoded?
There are a few options available when using HTML forms:

`application/x-www-form-urlencoded` is the most suitable to our usecase.
Quoting MDN web docs, with `application/x-www-form-urlencoded`

> the keys and values [in our form] are encoded in key-value tuples separated by '&', with a '=' between the key and the value. Non-alphanumeric characters in both keys and values are percent encoded.

For example: if the name is `Le Guin` and the email is `ursula_le_guin@gmail.com` the `POST` request body should be `name=le%20guin&email=ursula_le_guin%40gmail.com` (spaces are replaced by `%20` while `@` becomes `%40` - a reference conversion table can be found here).

To summarise:

- if a valid pair of name and email is supplied using the `application/x-www-form-urlencoded` format the backend should return a `200 OK`;
- if either name or email are missing the backend should return a `400 BAD REQUEST`.

3.7.2 Capturing Our Requirements As Tests

Now that we understand better what needs to happen, let's encode our expectations in a couple of integration tests.

Let's add the new tests to the existing `tests/health_check.rs` file - we will re-organise our test suite folder structure afterwards.

```rust
//! tests/health_check.rs
use std::net::TcpListener;

/// Spin up an instance of our application
/// and returns its address (i.e. http://localhost:XXXX)
fn spawn_app() -> String {
    [...]
}

#[tokio::test]
async fn health_check_works() {
    [...]
}

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app_address = spawn_app();
    let client = reqwest::Client::new();

    // Act
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    let response = client
        .post(&format!("{}/subscriptions", &app_address))
        .header("Content-Type", "application/x-www-form-urlencoded")
        .body(body)
        .send()
        .await
        .expect("Failed to execute request.");
    // Assert
    assert_eq!(200, response.status().as_u16());
}


#[tokio::test]
async fn subscribe_returns_a_400_when_data_is_missing() {
    // Arrange
    let app_address = spawn_app();
    let client = reqwest::Client::new();
    let test_cases = vec![
        ("name=le%20guin", "missing the email"),
        ("email=ursula_le_guin%40gmail.com", "missing the name"),
        ("", "missing both name and email")
    ];

    for (invalid_body, error_message) in test_cases {
        // Act
        let response = client
            .post(&format!("{}/subscriptions", &app_address))
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(invalid_body)
            .send()
            .await
            .expect("Failed to execute request.");

        // Assert
        assert_eq!(
            400,
            response.status().as_u16(),
            // Additional customised error message on test failure
            "The API did not fail with 400 Bad Request when the payload was {}.",
            error_message
        );
    }
}
```

`subscribe_returns_a_400_when_data_is_missing` is an example of *table-driven test* also known as *parametrised test*.
It is particularly helpful when dealing with bad inputs - instead of duplicating test logic several times we can simply run the same assertion against a collection of known invalid bodies that we expect to fail in the same way.
With parametrised tests it is important to have good error messages on failures: `assertion failed on line XYZ` is not great if you cannot tell which specific input is broken! On the flip side, that parametrised test is covering a lot of ground so it makes sense to invest a bit more time in generating a nice failure message.
Test frameworks in other languages sometimes have native support for this testing style (e.g. parametrised tests in `pytest` or `InlineData` in xUnit for C#) - there are a few crates in the Rust ecosystem that extend the basic test framework with similar features, but unfortunately they do not interop very well with the `#[tokio::test]` macro that we need to write asynchronous tests idiomatically (see `rstest` or `test-case`).

Let's run our test suite now:

```
---- health_check::subscribe_returns_a_200_for_valid_form_data stdout ----
thread 'health_check::subscribe_returns_a_200_for_valid_form_data'
panicked at 'assertion failed: `(left == right)`
  left: `200`,
 right: `404`:

---- health_check::subscribe_returns_a_400_when_data_is_missing stdout ----
thread 'health_check::subscribe_returns_a_400_when_data_is_missing'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `404`:
 The API did not fail with 400 Bad Request when the payload was missing the email.'
```

As expected, all our new tests are failing.
You can immediately spot a limitation of "roll-your-own" parametrised tests: as soon as one test case fails, the execution stops and we do not know the outcome for the following tests cases.

Let's get started on the implementation.

3.7.3 Parsing Form Data From A `POST` Request

All tests are failing because the application returns a `404 NOT FOUND` for `POST` requests hitting `/subscriptions`. Legitimate behaviour: we do not have a handler registered for that path.
Let's fix it by adding a matching route to our `App` in `src/lib.rs`:

```rust
//! src/lib.rs
use actix_web::dev::Server;
use actix_web::{web, App, HttpResponse, HttpServer};
use std::net::TcpListener;

// We were returning `impl Responder` at the very beginning.
// We are now spelling out the type explicitly given that we have
// become more familiar with `actix-web`.
// There is no performance difference! Just a stylistic choice :)
async fn health_check() -> HttpResponse {
    HttpResponse::Ok().finish()
}

// Let's start simple: we always return a 200 OK
async fn subscribe() -> HttpResponse {
    HttpResponse::Ok().finish()
}

pub fn run(listener: TcpListener) -> Result<Server, std::io::Error> {
    let server = HttpServer::new(|| {
        App::new()
            .route("/health_check", web::get().to(health_check))
            // A new entry in our routing table for POST /subscriptions requests
            .route("/subscriptions", web::post().to(subscribe))
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

Running our test suite again:

```
running 3 tests
test health_check::health_check_works ... ok
test health_check::subscribe_returns_a_200_for_valid_form_data ... ok
test health_check::subscribe_returns_a_400_when_data_is_missing ... FAILED

failures:

---- health_check::subscribe_returns_a_400_when_data_is_missing stdout ----
thread 'health_check::subscribe_returns_a_400_when_data_is_missing'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `200`:
 The API did not fail with 400 Bad Request when the payload was missing the email.'

failures:
    health_check::subscribe_returns_a_400_when_data_is_missing

test result: FAILED. 2 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

`subscribe_returns_a_200_for_valid_form_data` now passes: well, our handler accepts **all** incoming data as valid, no surprises there.
`subscribe_returns_a_400_when_data_is_missing`, instead, is still red.
Time to do some real parsing on that request body. What does `actix-web` offer us?

3.7.3.1 Extractors

Quite prominent on `actix-web`'s User Guide is the Extractors' section.
Extractors are used, as the name implies, to tell the framework to *extract* certain pieces of information from an incoming request.
`actix-web` provides several extractors out of the box to cater for the most common usecases:

- `Path` to get dynamic path segments from a request's path;
- `Query` for query parameters;
- `Json` to parse a JSON-encoded request body;
- etc.

Luckily enough, there is an extractor that serves exactly our usecase: `Form`.
Reading straight from its documentation:

> `Form` data helper (`application/x-www-form-urlencoded`).
> Can be used to extract url-encoded data from the request body, or send url-encoded data as the response.

That's music to my ears.
How do we use it?

Looking at `actix-web`'s User Guide:

> An extractor can be accessed as an argument to a handler function. Actix-web supports up to 10 extractors per handler function. Argument position does not matter.

Example:

```rust
use actix_web::web;

#[derive(serde::Deserialize)]
struct FormData {
    username: String,
}

/// Extract form data using serde.
/// This handler get called only if content type is *x-www-form-urlencoded*
/// and content of the request could be deserialized to a `FormData` struct
fn index(form: web::Form<FormData>) -> String {
    format!("Welcome {}!", form.username)
}
```

So, basically... you just slap it there as an argument of your handler and `actix-web`, when a request comes in, will somehow do the heavy-lifting for you. Let's ride along for now and we will circle back later to understand what is happening under the hood.

Our `subscribe` handler currently looks like this:

```rust
//! src/lib.rs
// Let's start simple: we always return a 200 OK
async fn subscribe() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

Using the example as a blueprint, we probably want something along these lines:

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

#[derive(serde::Deserialize)]
struct FormData {
    email: String,
    name: String
}

async fn subscribe(_form: web::Form<FormData>) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

`cargo check` is not happy:

```
error[E0433]: failed to resolve: use of undeclared type or module `serde`
 --> src/lib.rs:9:10
  |
9 | #[derive(serde::Deserialize)]
  |          ^^^^^ use of undeclared type or module `serde`
```

Fair enough: we need to add `serde` to our dependencies. Let's add a new line to our `Cargo.toml`:

```toml
[dependencies]
# We need the optional `derive` feature to use `serde`'s procedural macros:
# `#[derive(Serialize)]` and `#[derive(Deserialize)]`.
# The feature is not enabled by default to avoid pulling in
# unnecessary dependencies for projects that do not need it.
```

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

`cargo check` should succeed now. What about `cargo test`?

```
running 3 tests
test health_check_works ... ok
test subscribe_returns_a_200_for_valid_form_data ... ok
test subscribe_returns_a_400_when_data_is_missing ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

They are all green!

But *why*?

3.7.3.2 `Form` And `FromRequest`

Let's go straight to the source: what does `Form` look like?
You can find its source code here.

The definition seems fairly innocent:

```rust
#[derive(PartialEq, Eq, PartialOrd, Ord)]
pub struct Form<T>(pub T);
```

It is nothing more than a wrapper: it is generic over a type `T` which is then used to populate `Form`'s only field.
Not much to see here.
Where does the extraction magic take place?

An extractor is a type that implements the `FromRequest` trait.
`FromRequest`'s definition is a bit noisy because Rust does not yet support `async fn` in trait definitions. Reworking it slightly, it boils down to something that looks more or less like this:

```rust
/// Trait implemented by types that can be extracted from request.
///
/// Types that implement this trait can be used with `Route` handlers.
pub trait FromRequest: Sized {
    type Error = Into<actix_web::Error>;

    async fn from_request(
        req: &HttpRequest,
        payload: &mut Payload
    ) -> Result<Self, Self::Error>;

    /// Omitting some ancillary methods that actix-web implements
    /// out of the box for you and supporting associated types
    ///  [...]
}
```

`from_request` takes as inputs the head of the incoming HTTP request (i.e. `HttpRequest`) and the bytes of its payload (i.e. `Payload`). It then returns `Self`, if the extraction succeeds, or an error type that can be converted into `actix_web::Error`.
All arguments in the signature of a route handler must implement the `FromRequest` trait: `actix-web` will invoke `from_request` for each argument and, if the extraction succeeds for all of them, it will then run the actual handler function.

If one of the extractions fails, the corresponding error is returned to the caller and the handler is never invoked (`actix_web::Error` can be converted to a `HttpResponse`).

This is extremely convenient: your handler does not have to deal with the raw incoming request and can instead work directly with strongly-typed information, significantly simplifying the code that you need to write to handle a request.

Let's look at `Form`'s `FromRequest` implementation: what does it do?
Once again, I slightly reshaped the actual code to highlight the key elements and ignore the nitty-gritty implementation details.

```rust
impl<T> FromRequest for Form<T>
where
    T: DeserializeOwned + 'static,
{
    type Error = actix_web::Error;

    async fn from_request(
        req: &HttpRequest,
        payload: &mut Payload
    ) -> Result<Self, Self::Error> {
        // Omitted stuff around extractor configuration (e.g. payload size limits)

        match UrlEncoded::new(req, payload).await {
            Ok(item) => Ok(Form(item)),
            // The error handler can be customised.
            // The default one will return a 400, which is what we want.
            Err(e) => Err(error_handler(e))
        }
    }
}
```

All the heavy-lifting seems to be happening inside that `UrlEncoded` struct.
`UrlEncoded` does *a lot:* it transparently handles compressed and uncompressed payloads, it deals with the fact that the request body arrives a chunk at a time as a stream of bytes, etc.
The key passage, after all those things have been taken care of, is:

```rust
serde_urlencoded::from_bytes::<T>(&body).map_err(|_| UrlencodedError::Parse)
```

`serde_urlencoded` provides (de)serialisation support for the `application/x-www-form-urlencoded` data format.
`from_bytes` takes as input a contiguous slice of bytes and it deserialises an instance of type `T` from it according to the rules of the URL-encoded format: the keys and values are encoded in key-value tuples separated by `&`, with a `=` between the key and the value; non-alphanumeric characters in both keys and values are percent encoded.

How does it know how to do it for a generic type `T`?
It is because `T` implements the `DeserializedOwned` trait from `serde`:

```rust
impl<T> FromRequest for Form<T>
where
    T: DeserializeOwned + 'static,
{
// [...]
}
```

To understand what is *actually* going under the hood we need to take a closer look at `serde` itself.

> The next section on `serde` touches on a couple of advanced Rust topics.
> It's fine if not everything falls into place the first time you read it!
> Come back to it once you have played with Rust and `serde` a bit more to deep-dive on the toughest bits of it.

3.7.3.3 Serialisation In Rust: serde

Why do we need `serde`? What does `serde` actually *do* for us?
Quoting from its guide:

> Serde is a framework for **ser**ializing and **de**serializing Rust data structures efficiently and generically.

3.7.3.3.1 Generically `serde` does not, by itself, provide support for (de)serialisation from/to any specific data format: you will not find any code inside `serde` that deals with the specifics of JSON, Avro or MessagePack. If you need support for a specific data format, you need to pull in another crate (e.g. `serde_json` for JSON or `avro-rs` for Avro).
`serde` defines a set of *interfaces* or, as they themselves call it, a *data model*.

If you want to implement a library to support serialisation for a new data format, you have to provide an implementation of the `Serializer` trait.
Each method on the `Serializer` trait corresponds to one of the 29 types that form `serde`'s data model - your implementation of `Serializer` specifies how each of those types maps to your specific data format.
For example, if you were adding support for JSON serialisation, your `serialize_seq` implementation would output an opening square bracket `[` and return a type which can be used to serialize sequence elements.[7]

On the other side, you have the `Serialize` trait: your implementation of `Serialize::serialize` for a Rust type is meant to specify how to decompose it according to `serde`'s data model using the methods available on the `Serializer` trait.
Using again the sequence example, this is how `Serialize` is implemented for a Rust vector:

```rust
use serde::ser::{Serialize, Serializer, SerializeSeq};

impl<T> Serialize for Vec<T>
where
    T: Serialize,
{
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: Serializer,
    {
        let mut seq = serializer.serialize_seq(Some(self.len()))?;
        for element in self {
```

[7] You can look at `serde_json`'s `serialize_seq` implementation for confirmation: here. There is an optimisation for empty sequences (you immediately output `[]`), but that is pretty much what is happening.

```
            seq.serialize_element(element)?;
        }
        seq.end()
    }
}
```

That is what allows `serde` to be agnostic with respect to data formats: once your type implements `Serialize`, you are then free to use any concrete implementation of `Serializer` to actually perform the serialisation step - i.e. you can serialize your type to any format for which there is an available `Serializer` implementation on `crates.io` (*spoiler:* almost all commonly used data formats).
The same is true for deserialisation, via `Deserialize` and `Deserializer`, with a few additional details around lifetimes to support zero-copy deserialisation.

3.7.3.3.2 Efficiently

What about speed?
Is `serde` slower due to the fact that it is generic over the underlying data formats?

No, thanks to a process called *monomorphization*.
Every time a generic function is called with a concrete set of types, the Rust compiler will create a copy of the function body replacing the generic type parameters with the concrete types. This allows the compiler to optimize each instance of the function body with respect to the concrete types involved: the result is no different from what we would have achieved writing down separate functions for each type, without using generics or traits. In other words, we do not pay any runtime costs for using generics[8].

This concept is extremely powerful and it's often referred to as zero-cost abstraction: using higher-level language constructs results in the same machine code you would have obtained with uglier/more "hand-rolled" implementations. We can therefore write code that is easier to read for a human (as it's supposed be!) without having to compromise on the quality of the final artifact.

`serde` is also extremely careful when it comes to memory usage: the intermediate data model that we spoke about is *implicitly* defined via trait methods, there is no real intermediate serialised struct. If you want to learn more about it, Josh Mcguigan wrote an amazing deep-dive titled Understanding Serde.

It is also worth pointing out that all information required to (de)serialize a specific type for a specific data format are available at *compile-time*, there is no runtime overhead.
(De)serializers in other languages often leverage runtime reflection to fetch information about the type you want to (de)serialize (e.g. the list of their field names). Rust does not provide runtime reflection and everything has to be specified upfront.

3.7.3.3.3 Conveniently

This is where `#[derive(Serialize)]` and `#[derive(Deserialize)]` come into the picture.
You really do not want to spell out, manually, how serialisation should be performed for every single type defined in your project. It is tedious, error-prone and it takes time away from the application-specific logic that you are supposed to be focused on.

[8] At the same time, it must be said that writing a serializer that is specialised for a single data format and a single usecase (e.g. batch-serialisation) might give you a chance to leverage algorithmic choices that are not compatible with the *structure* of `serde`'s data model, meant to support several formats for a variety of usecases. An example in this vein would be `simd-json`.

Those two procedural macros, bundled with `serde` behind the `derive` feature flag, will parse the definition of your type and automatically generate for you the right `Serialize`/`Deserialize` implementation.

3.7.3.4 Putting Everything Together

Given everything we learned so far, let's take a second look at our `subscribe` handler:

```rust
#[derive(serde::Deserialize)]
pub struct FormData {
    email: String,
    name: String,
}

// Let's start simple: we always return a 200 OK
async fn subscribe(_form: web::Form<FormData>) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

We now have a good picture of what is happening:

- before calling `subscribe` `actix-web` invokes the `from_request` method for all `subscribe`'s input arguments: in our case, `Form::from_request`;
- `Form::from_request` tries to deserialise the body into `FormData` according to the rules of URL-encoding leveraging `serde_urlencoded` and the `Deserialize` implementation of `FormData`, automatically generated for us by `#[derive(serde::Deserialize)]`;
- if `Form::from_request` fails, a `400 BAD REQUEST` is returned to the caller. If it succeeds, `subscribe` is invoked and we return a `200 OK`.

Take a moment to be amazed: it looks so deceptively simple, yet there is **so much** going on in there - we are leaning heavily on Rust's strength as well as some of the most polished crates in its ecosystem.

3.8 Storing Data: Databases

Our `POST /subscriptions` endpoint passes our tests but its usefulness is fairly limited: we are not *storing* valid emails and names anywhere.
There is no permanent record of the information that we collected from our HTML form.
How do we fix it?

When we defined what Cloud-native stands for we listed some of the emergent behaviour that we expect to see in our system: in particular, we want to achieve high-availability while running in a fault-prone environment.
Our application is therefore forced to be **distributed** - there should be multiple instances of it running on multiple machines in order to survive hardware failures.

This has consequences when it comes to data persistence: we cannot rely on the filesystem of our host as a storage layer for incoming data.
Anything that we save on disk would only be available to one of the many replicas of our application[9]. Furthermore, it would probably disappear if the underlying host crashed.

[9] Unless we implement some kind of synchronisation protocol between our replicas, which would quickly turn

This explains why Cloud-native applications are usually stateless: their persistence needs are delegated to specialised external systems - **databases**.

3.8.1 Choosing A Database

What database should we use for our newsletter project?

I will lay down my personal rule-of-thumb, which might sound controversial:

> If you are uncertain about your persistence requirements, use a relational database.
> If you have no reason to expect **massive** scale, use PostgreSQL.

The offering when it comes to databases has exploded in the last twenty years.
From a data-model perspective, the NoSQL movement has brought us document-stores (e.g. MongoDB), key-value stores (e.g. AWS DynamoDB), graph databases (e.g. Neo4J), etc.
We have databases that use RAM as their primary storage (e.g. Redis).
We have databases that are optimised for analytical queries via columnar storage (e.g. AWS RedShift).

There is a world of possibilities and you should definitely leverage this richness when designing systems.
Nonetheless, it is much easier to design yourself into a corner by using a *specialised* data storage solution when you still do not have a clear picture of the data access patterns used by your application.
Relational databases are reasonably good as jack-of-all-trades: they will often be a good choice when building the first version of your application, supporting you along the way while you explore the constraints of your domain[10].

Even when it comes to relational databases there is plenty of choice.
Alongside classics like PostgreSQL and MySQL you will find some exciting new entries like AWS Aurora, Google Spanner and CockroachDB.
What do they all have in common?
They are built to *scale*. Way beyond what traditional SQL databases were supposed to be able to handle.
If scale is a problem of yours, by all means, take a look there. If it isn't, you do not need to take onboard the additional complexity.

This is how we end up with PostgreSQL: a battle-tested piece of technology, widely supported across all cloud providers if you need a managed offering, opensource, exhaustive documentation, easy to run locally and in CI via Docker, well-supported within the Rust ecosystem.

3.8.2 Choosing A Database Crate

As of August 2020, there are three top-of-mind options when it comes to interacting with PostgreSQL in a Rust project:

- `tokio-postgres`;

into a badly-written poor-man-copy of a database.

[10] Relational databases provide you with **transactions** - a powerful mechanism to handle partial failures and manage concurrent access to shared data. We will discuss transactions in greater detail in Chapter 7.

- `sqlx`;
- `diesel`.

All three are massively popular projects that have seen significant adoption with a fair share of production usage. How do you pick one?

It boils down to how you feel about three topics:

- compile-time safety;
- SQL-first vs a DSL for query building;
- async vs sync interface.

3.8.2.1 Compile-time Safety

When interacting with a relational database it is fairly easy to make mistakes - we might, for example,

- have a typo in the name of a column or a table mentioned in our query;
- try to perform operations that are rejected by the database engine (e.g. summing a string and a number or joining two tables on the wrong column);
- expect to have a certain field in the returned data that is actually not there.

The key question is: **when** do we realise we made a mistake?

In most programming languages, it will be **at runtime**: when we try to execute our query the database will reject it and we will get an error or an exception. This is what happens when using `tokio-postgres`.

`diesel` and `sqlx` try to speed up the feedback cycle by detecting **at compile-time** most of these mistakes.
`diesel` leverages its CLI to generate a representation of the database schema as Rust code, which is then used to check assumptions on all of your queries.
`sqlx`, instead, uses procedural macros to connect to a database at compile-time and check if the provided query is indeed sound[11].

3.8.2.2 Query Interface

Both `tokio-postgres` and `sqlx` expect you to use SQL directly to write your queries.

`diesel`, instead, provides its own query builder: queries are represented as Rust types and you add filters, perform joins and similar operations by calling methods on them. This is often referred to with the name of **D**omain **S**pecific **L**anguage (DSL).

Which one is better?
As always, it depends.

SQL is extremely portable - you can use it in any project where you have to interact with a relational database, regardless of the programming language or the framework the application is written with.
`diesel`'s DSL, instead, is only relevant when working with `diesel`: you have to pay an upfront

[11] Performing IO in a procedural macro is somewhat controversial and forces you to always have a database up and running when working on a `sqlx` project; `sqlx` is adding support for "offline" builds by caching the retrieved query metadata in its upcoming `0.4.0` release.

learning cost to become fluent with it and it only pays off if you stick to `diesel` for your current and future projects. It is also worth pointing out that expressing complex queries using `diesel`'s DSL can be difficult and you might end up having to write raw SQL anyway.

On the flip side, `diesel`'s DSL makes it easier to write reusable components: you can split your complex queries into smaller units and leverage them in multiple places, as you would do with a normal Rust function.

3.8.2.3 Async Support

I remember reading somewhere a killer explanation of async IO that more or less sounded like this:

> Threads are for working in parallel, async is for waiting in parallel.

Your database is not sitting next to your application on the same physical machine host: to run queries you have to perform network calls.
An asynchronous database driver will not reduce how long it takes to process a single query, but it will enable your application to leverage all CPU cores to perform other meaningful work (e.g. serve another HTTP request) while waiting for the database to return results.

Is this a significant enough benefit to take onboard the additional complexity introduced by asynchronous code?
It depends on the performance requirements of your application.
Generally speaking, running queries on a separate threadpool should be more than enough for most usecases. At the same time, if your web framework is already asynchronous, using an asynchronous database driver will actually give you less headaches[12].

Both `sqlx` and `tokio-postgres` provide an asynchronous interface, while `diesel` is synchronous and does not plan to roll out async support in the near future.

It is also worth mentioning that `tokio-postgres` is, at the moment, the only crate that supports query pipelining. The feature is still at the design stage for `sqlx` while I could not find it mentioned anywhere in `diesel`'s docs or issue tracker.

3.8.2.4 Summary

Let's summarise everything we covered in a comparison matrix:

Crate	Compile-time safety	Query interface	Async
`tokio-postgres`	No	SQL	Yes
`sqlx`	Yes	SQL	Yes
`diesel`	Yes	DSL	No

[12] Async runtimes are based around the assumptions that futures, when polled, will yield control back to the executor "very quickly". If you run blocking IO code by mistake on the same threadpool used by the runtime to poll asynchronous tasks you get yourself in troubles - e.g. your application might mysteriously hang under load. You have to be careful and always make sure that blocking IO is performed on a separate threadpool using functions like `tokio::spawn_blocking` or `async_std::spawn_blocking`.

3.8.2.5 Our Pick: `sqlx`

For *Zero To Production* we will use `sqlx`: its asynchronous support simplifies the integration with `actix-web` without forcing us to compromise on compile-time guarantees. It also limits the API surface that we have to cover and become proficient with thanks to its usage of raw SQL for queries.

3.8.3 Integration Testing With Side-effects

What do we want to accomplish?
Let's look again at our "happy case" test:

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

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app_address = spawn_app();
    let client = reqwest::Client::new();

    // Act
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    let response = client
        .post(&format!("{}/subscriptions", &app_address))
        .header("Content-Type", "application/x-www-form-urlencoded")
        .body(body)
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(200, response.status().as_u16());
}
```

The assertion we have there is not enough.
We have no way to tell, just by looking at the API response, if the desired business outcome has been achieved - we are interested to know if a *side-effect* has taken place, i.e. data storage.

We want to check if the details of our new subscriber have actually been persisted.
How do we go about it?

We have two options:

1. leverage another endpoint of our public API to inspect the application state;
2. query directly the database in our test case.

Option 1 should be your go-to when possible: your tests remain oblivious to the implementation details of the API (e.g. the underlying database technology and its schema) and are therefore less likely to be disrupted by future refactorings.

Unfortunately we do not have any public endpoint on our API that allows us to verify if a subscriber exists.
We could add a `GET /subscriptions` endpoint to fetch the list of existing subscribers, but we would then have to worry about securing it: we do not want to have the names and emails of our subscribers exposed on the public internet without any form of authentication.

We will probably end up writing a `GET /subscriptions` endpoint down the line (i.e. we do not want to log into our production database to check the list of our subscribers), but we should not start writing a new feature just to test the one we are working on.

Let's bite the bullet and write a small query in our test. We will remove it down the line when a better testing strategy becomes available.

3.8.4 Database Setup

To run queries in our test suite we need:

- a running Postgres instance[13];
- a table to store our subscribers data.

3.8.4.1 Docker

To run Postgres we will use Docker - before launching our test suite we will launch a new Docker container using Postgres' official Docker image.
You can follow the instructions on Docker's website to install it on your machine.

Let's create a small bash script for it, `scripts/init_db.sh`, with a few knobs to customise Postgres' default settings:

```bash
#!/usr/bin/env bash
set -x
set -eo pipefail

# Check if a custom user has been set, otherwise default to 'postgres'
DB_USER=${POSTGRES_USER:=postgres}
# Check if a custom password has been set, otherwise default to 'password'
DB_PASSWORD="${POSTGRES_PASSWORD:=password}"
# Check if a custom database name has been set, otherwise default to 'newsletter'
DB_NAME="${POSTGRES_DB:=newsletter}"
# Check if a custom port has been set, otherwise default to '5432'
DB_PORT="${POSTGRES_PORT:=5432}"

# Launch postgres using Docker
docker run \
  -e POSTGRES_USER=${DB_USER} \
  -e POSTGRES_PASSWORD=${DB_PASSWORD} \
  -e POSTGRES_DB=${DB_NAME} \
  -p "${DB_PORT}":5432 \
  -d postgres \
  postgres -N 1000
  # ^ Increased maximum number of connections for testing purposes
```

Let's make it executable:

```bash
chmod +x scripts/init_db.sh
```

We can then launch Postgres with

[13] I do not belong to the "in-memory test database" school of thought: whenever possible you should strive to use the same database both for your tests and your production environment. I have been burned one time too many by differences between the in-memory stub and the real database engine to believe it provides any kind of benefit over using "the real thing".

```
./scripts/init_db.sh
```

If you run `docker ps` you should see something along the lines of

```
IMAGE            PORTS                       STATUS
postgres   127.0.0.1:5432->5432/tcp   Up 12 seconds    [ ... ]
```

N.B. - the port mapping bit could be slightly different if you are not using Linux!

3.8.4.2 Database Migrations

To store our subscribers details we need to create our first table.
To add a new table to our database we need to change its schema - this is commonly referred to as a *database migration*.

3.8.4.2.1 `sqlx-cli`

`sqlx` provides a command-line interface, `sqlx-cli`, to manage database migrations.

We can install the CLI with

```
cargo install --version=0.5.7 sqlx-cli --no-default-features --features postgres
```

Run `sqlx --help` to check that everything is working as expected.

3.8.4.2.2 Database Creation

The first command we will usually want to run is `sqlx database create`. According to the help docs:

```
sqlx-database-create
Creates the database specified in your DATABASE_URL

USAGE:
    sqlx database create

FLAGS:
    -h, --help       Prints help information
    -V, --version    Prints version information
```

In our case, this is not strictly necessary: our Postgres Docker instance already comes with a default database named `newsletter`, thanks to the settings we specified when launching it using environment variables. Nonetheless, you will have to go through the creation step in your CI pipeline and in your production environment, so worth covering it anyway.

As the help docs imply, `sqlx database create` relies on the `DATABASE_URL` environment variable to know what to do.
`DATABASE_URL` is expected to be a valid Postgres connection string - the format is as follows:

```
postgres://${DB_USER}:${DB_PASSWORD}@${DB_HOST}:${DB_PORT}/${DB_NAME}
```

We can therefore add a couple more lines to our `scripts/init_db.sh` script[14]:

```
# [ ... ]

```

[14] If you run the script again now it will fail because there is a Docker container with same name already running! You have to stop/kill it before running the updated version of the script.

```bash
export DATABASE_URL=postgres://${DB_USER}:${DB_PASSWORD}@localhost:${DB_PORT}/${DB_NAME}
sqlx database create
```

You might run into an annoying issue from time to time: the Postgres container will not be ready to accept connections when we try to run `sqlx database create`.
It happened to me often enough to look for a workaround: we need to wait for Postgres to be healthy before starting to run commands against it. Let's update our script to:

```bash
#!/usr/bin/env bash
set -x
set -eo pipefail

DB_USER=${POSTGRES_USER:=postgres}
DB_PASSWORD="${POSTGRES_PASSWORD:=password}"
DB_NAME="${POSTGRES_DB:=newsletter}"
DB_PORT="${POSTGRES_PORT:=5432}"

docker run \
  -e POSTGRES_USER=${DB_USER} \
  -e POSTGRES_PASSWORD=${DB_PASSWORD} \
  -e POSTGRES_DB=${DB_NAME} \
  -p "${DB_PORT}":5432 \
  -d postgres \
  postgres -N 1000

# Keep pinging Postgres until it's ready to accept commands
export PGPASSWORD="${DB_PASSWORD}"
until psql -h "localhost" -U "${DB_USER}" -p "${DB_PORT}" -d "postgres" -c '\q'; do
  >&2 echo "Postgres is still unavailable - sleeping"
  sleep 1
done

>&2 echo "Postgres is up and running on port ${DB_PORT}!"

export DATABASE_URL=postgres://${DB_USER}:${DB_PASSWORD}@localhost:${DB_PORT}/${DB_NAME}
sqlx database create
```

Problem solved!
The health check uses `psql`, the command line client for Postgres. Check these instructions on how to install it on your OS.

Scripts do not come bundled with a manifest to declare their dependencies: it's unfortunately very common to launch a script without having installed all the prerequisites. This will usually result in the script crashing mid-execution, sometimes leaving stuff in our system in a half-broken state.
We can do better in our initialization script: let's check that both `psql` and `sqlx-cli` are installed at the very beginning.

```bash
set -x
set -eo pipefail

if ! [ -x "$(command -v psql)" ]; then
  echo >&2 "Error: psql is not installed."
  exit 1
fi
```

```
if ! [ -x "$(command -v sqlx)" ]; then
  echo >&2 "Error: sqlx is not installed."
  echo >&2 "Use:"
  echo >&2 "    cargo install --version=0.5.7 sqlx-cli --no-default-features --features postgres"
  echo >&2 "to install it."
  exit 1
fi

# The rest of the script
```

3.8.4.2.3 Adding A Migration

Let's create our first migration now with

```
# Assuming you used the default parameters to launch Postgres in Docker!
export DATABASE_URL=postgres://postgres:password@127.0.0.1:5432/newsletter
sqlx migrate add create_subscriptions_table
```

A new top-level directory should have now appeared in your project - `migrations`. This is where all migrations for our project will be stored by `sqlx`'s CLI.
Under `migrations` you should already have one file called `{timestamp}_create_subscriptions_table.sql` - this is where we have to write the SQL code for our first migration.

Let's quickly sketch the query we need:

```
-- migrations/{timestamp}_create_subscriptions_table.sql
-- Create Subscriptions Table
CREATE TABLE subscriptions(
   id uuid NOT NULL,
   PRIMARY KEY (id),
   email TEXT NOT NULL UNIQUE,
   name TEXT NOT NULL,
   subscribed_at timestamptz NOT NULL
);
```

There is a endless debate when it comes to primary keys: some people prefer to use columns with a business meaning (e.g. `email`, a *natural key*), others feel safer with a synthetic key without any business meaning (e.g. `id`, a randomly generated UUID, a *surrogate key*).
I generally default to a synthetic identifier unless I have a very compelling reason not to - feel free to disagree with me here.

A couple of other things to make a note of:

- we are keeping track of when a subscription is created with `subscribed_at` (`timestamptz` is a time-zone aware date and time type);
- we are enforcing email uniqueness at the database-level with a `UNIQUE` constraint;
- we are enforcing that all fields should be populated with a `NOT NULL` constraint on each column;
- we are using `TEXT` for `email` and `name` because we do not have any restriction on their maximum lengths.

Database constraints are useful as a last line of defence from application bugs but they come at a cost - the database has to ensure all checks pass before writing new data into the table. Therefore constraints impact our write-throughput, i.e. the number of rows we can `INSERT/UPDATE` per unit of time in a table.

`UNIQUE`, in particular, introduces an additional B-tree index on our `email` column: the index has to be updated on every `INSERT`/`UPDATE`/`DELETE` query and it takes space on disk.

In our specific case, I would not be too worried: our mailing list would have to be ***incredibly popular*** for us to encounter issues with our write throughput. Definitely a good problem to have, if it comes to that.

3.8.4.2.4 Running Migrations

We can run migrations against our database with

```
sqlx migrate run
```

It has the same behaviour of `sqlx database create` - it will look at the `DATABASE_URL` environment variable to understand what database needs to be migrated.

Let's add it to our `scripts/init_db.sh` script:

```
#!/usr/bin/env bash
set -x
set -eo pipefail

if ! [ -x "$(command -v psql)" ]; then
  echo >&2 "Error: psql is not installed."
  exit 1
fi

if ! [ -x "$(command -v sqlx)" ]; then
  echo >&2 "Error: sqlx is not installed."
  echo >&2 "Use:"
  echo >&2 "    cargo install --version=0.5.7 sqlx-cli --no-default-features --features postgres"
  echo >&2 "to install it."
  exit 1
fi

DB_USER=${POSTGRES_USER:=postgres}
DB_PASSWORD="${POSTGRES_PASSWORD:=password}"
DB_NAME="${POSTGRES_DB:=newsletter}"
DB_PORT="${POSTGRES_PORT:=5432}"

# Allow to skip Docker if a dockerized Postgres database is already running
if [[ -z "${SKIP_DOCKER}" ]]
then
  docker run \
      -e POSTGRES_USER=${DB_USER} \
      -e POSTGRES_PASSWORD=${DB_PASSWORD} \
      -e POSTGRES_DB=${DB_NAME} \
      -p "${DB_PORT}":5432 \
      -d postgres \
      postgres -N 1000
fi

export PGPASSWORD="${DB_PASSWORD}"
until psql -h "localhost" -U "${DB_USER}" -p "${DB_PORT}" -d "postgres" -c '\q'; do
  >&2 echo "Postgres is still unavailable - sleeping"
  sleep 1
done
```

```
>&2 echo "Postgres is up and running on port ${DB_PORT} - running migrations now!"

export DATABASE_URL=postgres://${DB_USER}:${DB_PASSWORD}@localhost:${DB_PORT}/${DB_NAME}
sqlx database create
sqlx migrate run

>&2 echo "Postgres has been migrated, ready to go!"
```

We have put the `docker run` command behind a `SKIP_DOCKER` flag to make it easy to run migrations against an existing Postgres instance without having to tear it down manually and re-create it with `scripts/init_db.sh`. It will also be useful in CI, if Postgres is not spun up by our script.

We can now migrate the database with

```
SKIP_DOCKER=true ./scripts/init_db.sh
```

You should be able to spot, in the output, something like

```
+ sqlx migrate run
20200823135036/migrate create subscriptions table (7.563944ms)
```

If you check your database using your favourite graphic interface for Postgres you will now see a `subscriptions` table alongside a brand new `_sqlx_migrations` table: this is where `sqlx` keeps track of what migrations have been run against your database - it should contain a single row now for our `create_subscriptions_table` migration.

3.8.5 Writing Our First Query

We have a migrated database up and running. How do we talk to it?

3.8.5.1 Sqlx Feature Flags

We installed `sqlx-cli`, but we have actually not yet added `sqlx` itself as a dependency of our application.
Let's append a new line to our `Cargo.toml`:

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

# Using table-like toml syntax to avoid a super-long line!
[dependencies.sqlx]
version = "0.5.7"
default-features = false
features = [
    "runtime-actix-rustls",
    "macros",
    "postgres",
    "uuid",
    "chrono",
    "migrate"
]
```

Yeah, there are a lot of feature flags. Let's go through all of them one by one:

- `runtime-actix-rustls` tells `sqlx` to use the `actix` runtime for its futures and `rustls` as TLS backend;
- `macros` gives us access to `sqlx::query!` and `sqlx::query_as!`, which we will be using extensively;
- `postgres` unlocks Postgres-specific functionality (e.g. non-standard SQL types);
- `uuid` adds support for mapping SQL UUIDs to the `Uuid` type from the `uuid` crate. We need it to work with our `id` column;
- `chrono` adds support for mapping SQL `timestamptz` to the `DateTime<T>` type from the `chrono` crate. We need it to work with our `subscribed_at` column;
- `migrate` gives us access to the same functions used under the hood by `sqlx-cli` to manage migrations. It will turn out to be useful for our test suite.

These should be enough for what we need to do in this chapter.

3.8.5.2 Configuration Management

The simplest entrypoint to connect to a Postgres database is `PgConnection`.
`PgConnection` implements the `Connection` trait which provides us with a `connect` method: it takes as input a connection string and returns us, asynchronously, a `Result<PostgresConnection, sqlx::Error>`.

Where do we get a connection string?

We could hard-code one in our application and then use it for our tests as well.
Or we could choose to introduce immediately some basic mechanism of configuration management.

It is simpler than it sounds and it will save us the cost of tracking down a bunch of hard-coded values across the whole application.
The `config` crate is Rust's swiss-army knife when it comes to configuration: it supports multiple file formats and it lets you combine different sources hierarchically (e.g. environment variables, configuration files, etc.) to easily customise the behaviour of your application for each deployment environment.

We do not need anything fancy for the time being: a single configuration file will do.

3.8.5.2.1 Making Space Right now all our application code lives in a single file, `lib.rs`.
Let's quickly split it into multiple sub-modules to avoid chaos now that we are adding new functionality. We want to land on this folder structure:

```
src/
  configuration.rs
  lib.rs
  main.rs
  routes/
    mod.rs
    health_check.rs
    subscriptions.rs
  startup.rs
```

Our `lib.rs` file becomes

```rust
//! src/lib.rs
pub mod configuration;
pub mod routes;
pub mod startup;
```

`startup.rs` will host our `run` function, `health_check` goes into `routes/health_check.rs`, `subscribe` and `FormData` into `routes/subscriptions.rs`, `configuration.rs` starts empty. Both handlers are re-exported in `routes/mod.rs`:

```rust
//! src/routes/mod.rs
mod health_check;
mod subscriptions;

pub use health_check::*;
pub use subscriptions::*;
```

You might have to add a few `pub` visibility modifiers here and there, as well as performing a few corrections to `use` statements in `main.rs` and `tests/health_check.rs`.

Make sure `cargo test` comes out green before moving forward.

3.8.5.2.2 Reading A Configuration File

To manage configuration with `config` we must represent our application settings as a Rust type that implements `serde`'s `Deserialize` trait.
Let's create a new `Settings` struct:

```rust
//! src/configuration.rs
#[derive(serde::Deserialize)]
pub struct Settings {}
```

We have two groups of configuration values at the moment:

- the application port, where `actix-web` is listening for incoming requests (currently hard-coded to `8000` in `main.rs`);
- the database connection parameters.

Let's add a field for each of them to `Settings`:

```rust
//! src/configuration.rs
#[derive(serde::Deserialize)]
pub struct Settings {
    pub database: DatabaseSettings,
    pub application_port: u16
}

#[derive(serde::Deserialize)]
pub struct DatabaseSettings {
    pub username: String,
    pub password: String,
    pub port: u16,
    pub host: String,
    pub database_name: String,
}
```

We need `#[derive(serde::Deserialize)]` on top of `DatabaseSettings` otherwise the compiler will complain with

```
error[E0277]: the trait bound
`configuration::DatabaseSettings: configuration::_::_serde::Deserialize<'_>`
is not satisfied
 --> src/configuration.rs:3:5
  |
3 |     pub database: DatabaseSettings,
  |     ^^^ the trait `configuration::_::_serde::Deserialize<'_>`
  |         is not implemented for `configuration::DatabaseSettings`
  |
  = note: required by `configuration::_::_serde::de::SeqAccess::next_element`
```

It makes sense: all fields in a type have to be deserialisable in order for the type as a whole to be deserialisable.

We have our configuration type, what now?

First of all, let's add `config` to our dependencies with

```
#! Cargo.toml
# [...]
[dependencies]
config = "0.11"
# [...]
```

We want to read our application settings from a configuration file named `configuration`:

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

pub fn get_configuration() -> Result<Settings, config::ConfigError> {
    // Initialise our configuration reader
    let mut settings = config::Config::default();

    // Add configuration values from a file named `configuration`.
    // It will look for any top-level file with an extension
    // that `config` knows how to parse: yaml, json, etc.
    settings.merge(config::File::with_name("configuration"))?;

    // Try to convert the configuration values it read into
    // our Settings type
    settings.try_into()
}
```

Let's modify our `main` function to read configuration as its first step:

```
//! src/main.rs
use std::net::TcpListener;
use zero2prod::startup::run;
use zero2prod::configuration::get_configuration;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // Panic if we can't read configuration
    let configuration = get_configuration().expect("Failed to read configuration.");
    // We have removed the hard-coded `8000` - it's now coming from our settings!
    let address = format!("127.0.0.1:{}", configuration.application_port);
    let listener = TcpListener::bind(address)?;
    run(listener)?.await
}
```

If you try to launch the application with `cargo run` it should crash:

```
Running `target/debug/zero2prod`

thread 'main' panicked at 'Failed to read configuration.:
configuration file "configuration" not found', src/main.rs:7:25

note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Panic in Arbiter thread.
```

Let's fix it by adding a configuration file.
We can use any file format for it, as long as `config` knows how to deal with it: we will go for YAML.

```yaml
# configuration.yaml
application_port: 8000
database:
  host: "127.0.0.1"
  port: 5432
  username: "postgres"
  password: "password"
  database_name: "newsletter"
```

`cargo run` should now execute smoothly.

3.8.5.3 Connecting To Postgres

`PgConnection::connect` wants a single connection string as input, while `DatabaseSettings` provides us with granular access to all the connection parameters. Let's add a convenient `connection_string` method to do it:

```rust
//! src/configuration.rs
// [...]
impl DatabaseSettings {
    pub fn connection_string(&self) -> String {
        format!(
            "postgres://{}:{}@{}:{}/{}",
            self.username, self.password, self.host, self.port, self.database_name
        )
    }
}
```

We are finally ready to connect!
Let's tweak our happy case test:

```rust
//! tests/health_check.rs
use sqlx::{PgConnection, Connection};
use zero2prod::configuration::get_configuration;
// [...]

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app_address = spawn_app();
    let configuration = get_configuration().expect("Failed to read configuration");
    let connection_string = configuration.database.connection_string();
    // The `Connection` trait MUST be in scope for us to invoke
    // `PgConnection::connect` - it is not an inherent method of the struct!
    let connection = PgConnection::connect(&connection_string)
        .await
        .expect("Failed to connect to Postgres.");
    let client = reqwest::Client::new();

    // Act
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    let response = client
        .post(&format!("{}/subscriptions", &app_address))
        .header("Content-Type", "application/x-www-form-urlencoded")
        .body(body)
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(200, response.status().as_u16());
}
```

And... `cargo test` works!
We just confirmed that we can successfully connect to Postgres from our tests!
A small step for the world, a huge leap forward for us.

3.8.5.4 Our Test Assertion

Now that we are connected, we can finally write the test assertions we have been dreaming about for the past 10 pages.
We will use `sqlx`'s `query!` macro:

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

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // [...]
    // The connection has to be marked as mutable!
    let mut connection = ...

    // Assert
    assert_eq!(200, response.status().as_u16());

    let saved = sqlx::query!("SELECT email, name FROM subscriptions",)
        .fetch_one(&mut connection)
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
}
```

What is the type of `saved`? The `query!` macro returns an anonymous record type: a struct definition is generated at compile-time after having verified that the query is valid, with a member for each column on the result (i.e. `saved.email` for the `email` column).

If we try to run `cargo test` we will get an error:

```
error: `DATABASE_URL` must be set to use query macros
  --> tests/health_check.rs:59:17
   |
59 |     let saved = sqlx::query!("SELECT email, name FROM subscriptions",)
   |                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: this error originates in a macro (in Nightly builds,
     run with -Z macro-backtrace for more info)
```

As we discussed before, `sqlx` reaches out to Postgres at compile-time to check that queries are well-formed. Just like `sqlx-cli` commands, it relies on the `DATABASE_URL` environment variable to know where to find the database.
We could export `DATABASE_URL` manually, but we would then run in the same issue every time we boot our machine and start working on this project. Let's take the advice of `sqlx`'s authors - we'll add a top-level `.env` file

```
DATABASE_URL="postgres://postgres:password@localhost:5432/newsletter"
```

`sqlx` will read `DATABASE_URL` from it and save us the hassle of re-exporting the environment variable every single time.
It feels a bit dirty to have the database connection parameters in two places (`.env` and `configuration.yaml`), but it is not a major problem: `configuration.yaml` can be used to alter the runtime behaviour of the application *after* it has been compiled, while `.env` is only relevant for our development process, build and test steps.
Commit the `.env` file to version control - we will need it in CI soon enough!

Let's try to run `cargo test` again:

```
running 3 tests
test health_check_works ... ok
test subscribe_returns_a_400_when_data_is_missing ... ok
test subscribe_returns_a_200_for_valid_form_data ... FAILED

failures:

---- subscribe_returns_a_200_for_valid_form_data stdout ----
thread 'subscribe_returns_a_200_for_valid_form_data' panicked at
'Failed to fetch saved subscription.: RowNotFound', tests/health_check.rs:59:17

failures:
    subscribe_returns_a_200_for_valid_form_data
```

It failed, which is exactly what we wanted!
We can now focus on patching the application to turn it green.

3.8.5.5 Updating Our CI Pipeline

If you check on it, you will notice that your CI pipeline is now failing to perform most of the checks we introduced at the beginning of our journey.
Our tests now rely on a running Postgres database to be executed properly. All our build commands (`cargo check`, `cargo lint`, `cargo build`), due to `sqlx`'s compile-time checks, need an

up-and-running database!

We do not want to venture further with a broken CI.
You can find an updated version of the GitHub Actions setup here. Only `general.yml` needs to be updated.

3.9 Persisting A New Subscriber

Just as we wrote a `SELECT` query to inspect what subscriptions had been persisted to the database in our test, we now need to write an `INSERT` query to actually store the details of a new subscriber when we receive a valid `POST /subscriptions` request.

Let's have a look at our request handler:

```rust
//! src/routes/subscriptions.rs
use actix_web::{web, HttpResponse};

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

// Let's start simple: we always return a 200 OK
pub async fn subscribe(_form: web::Form<FormData>) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

To execute a query within `subscribe` we need to get our hands on a database connection.
Let's figure out how to get one.

3.9.1 Application State In `actix-web`

So far our application has been entirely stateless: our handlers work solely with the data from the incoming request.
`actix-web` gives us the possibility to attach to the application other pieces of data that are not related to the lifecycle of a single incoming request - the so-called *application state*.

You can add information to the application state using the `app_data` method on `App`.

Let's try to use `app_data` to register a `PgConnection` as part of our application state. We need to modify our `run` method to accept a `PgConnection` alongside the `TcpListener`:

```rust
//! src/startup.rs

use crate::routes::{health_check, subscribe};
use actix_web::dev::Server;
use actix_web::{web, App, HttpServer};
use sqlx::PgConnection;
use std::net::TcpListener;

pub fn run(
    listener: TcpListener,
    // New parameter!
    connection: PgConnection
) -> Result<Server, std::io::Error> {
    let server = HttpServer::new(|| {
        App::new()
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            // Register the connection as part of the application state
            .app_data(connection)
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

`cargo check` is screaming at us:

```
error[E0277]: the trait bound `PgConnection: std::clone::Clone`
is not satisfied in `[closure@src/startup.rs:8:34: 13:6 PgConnection]`
  --> src/startup.rs:8:18
   |
8  |        let server = HttpServer::new(|| {
   |   __________________^^^^^^^^^^^^^^^_-
   |  |                  |
   |  |                  within `[closure@src/startup.rs:8:34: 13:6 PgConnection]`,
   |  |                  the trait `std::clone::Clone` is not implemented
   |  |                  for `PgConnection`
9  |  |         App::new()
10 |  |             .route("/health_check", web::get().to(health_check))
11 |  |             .route("/subscriptions", web::post().to(subscribe))
12 |  |             .app_data(connection)
13 |  |     })
   |  |_____- within this `[closure@src/startup.rs:8:34: 13:6 PgConnection]`
   |
   = note: required because it appears within the type
           `[closure@src/startup.rs:8:34: 13:6 PgConnection]`
   = note: required by `actix_web::server::HttpServer::<F, I, S, B>::new`

error[E0277]: the trait bound `PgConnection: std::clone::Clone`
is not satisfied in `[closure@src/startup.rs:8:34: 13:6 PgConnection]`
  --> src/startup.rs:8:18
   |
8  |         let server = HttpServer::new(|| {
   |    __________________^_______________-
   |   |__________________|
   |  ||
9  |  ||         App::new()
10 |  ||             .route("/health_check", web::get().to(health_check))
11 |  ||             .route("/subscriptions", web::post().to(subscribe))
12 |  ||             .app_data(connection)
13 |  ||     })
   |  ||_____- within this `[closure@src/startup.rs:8:34: 13:6 ::PgConnection]`
14 |  |      .listen(listener)?
   |  |_______________________^
   |    within `[closure@src/startup.rs:8:34: 13:6 ::PgConnection]`,
   |    the trait `std::clone::Clone` is not implemented for `PgConnection`
   |
   |
56 |         F: Fn() -> I + Send + Clone + 'static,
   |                              -----
   |    required by this bound in `actix_web::server::HttpServer`
   |
   = note: required because it appears within the type
           `[closure@src/startup.rs:8:34: 13:6 PgConnection]`
```

`HttpServer` expects `PgConnection` to be cloneable, which unfortunately is not the case.
Why does it need to implement `Clone` in the first place though?

3.9.2 `actix-web` Workers

Let's zoom in on our invocation of `HttpServer::new`:

```rust
let server = HttpServer::new(|| {
    App::new()
        .route("/health_check", web::get().to(health_check))
        .route("/subscriptions", web::post().to(subscribe))
})
```

`HttpServer::new` does not take `App` as argument - it wants *a closure that returns an* `App` *struct*.
This is to support `actix-web`'s runtime model: `actix-web` will spin up a worker process for each available core on your machine.
Each worker runs its own copy of the application built by `HttpServer` calling the very same closure that `HttpServer::new` takes as argument.

That is why `connection` has to be cloneable - we need to have one for every copy of `App`.
But, as we said, `PgConnection` does not implement `Clone` because it sits on top of a non-cloneable system resource, a TCP connection with Postgres. What do we do?

We can use `web::Data`, another `actix-web` extractor.
`web::Data` wraps our connection in an **A**tomic **R**eference **C**ounted pointer, an `Arc`: each instance of the application, instead of getting a raw copy of a `PgConnection`, will get a pointer to one.
`Arc<T>` is always cloneable, no matter who `T` is: cloning an `Arc` increments the number of active references and hands over a new copy of the memory address of the wrapped value.

Handlers can then access the application state using the same extractor.

Let's give it a try:

```rust
//! src/startup.rs
use crate::routes::{health_check, subscribe};
use actix_web::dev::Server;
use actix_web::{web, App, HttpServer};
use sqlx::PgConnection;
use std::net::TcpListener;

pub fn run(
    listener: TcpListener,
    connection: PgConnection
) -> Result<Server, std::io::Error> {
    // Wrap the connection in a smart pointer
    let connection = web::Data::new(connection);
    // Capture `connection` from the surrounding environment
    let server = HttpServer::new(move || {
        App::new()
```

```
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            // Get a pointer copy and attach it to the application state
            .app_data(connection.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

It doesn't compile *yet*, but we just need to do a bit of house-keeping:

```
error[E0061]: this function takes 2 arguments but 1 argument was supplied
  --> src/main.rs:11:5
   |
11 |     run(listener)?.await
   |     ^^^ -------- supplied 1 argument
   |     |
   |     expected 2 arguments
```

Let's fix the issue real quick:

```
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use sqlx::{Connection, PgConnection};
use std::net::TcpListener;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection = PgConnection::connect(&configuration.database.connection_string())
        .await
        .expect("Failed to connect to Postgres.");
    let address = format!("127.0.0.1:{}", configuration.application_port);
    let listener = TcpListener::bind(address)?;
    run(listener, connection)?.await
}
```

Perfect, it compiles.

3.9.3 The `Data` Extractor

We can now get our hands on an `Arc<PgConnection>` in our request handler, `subscribe`, using the `web::Data` extractor:

```
//! src/routes/subscriptions.rs
use sqlx::PgConnection;
// [...]

pub async fn subscribe(
    _form: web::Form<FormData>,
    // Retrieving a connection from the application state!
    _connection: web::Data<PgConnection>,
) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

We called `Data` an extractor, but what is it extracting a `PgConnection` from?
`actix-web` uses a *type-map* to represent its application state: a `HashMap` that stores arbitrary data (using the `Any` type) against their unique type identifier (obtained via `TypeId::of`).

`web::Data`, when a new request comes in, computes the `TypeId` of the type you specified in the signature (in our case `PgConnection`) and checks if there is a record corresponding to it in the type-map. If there is one, it casts the retrieved `Any` value to the type you specified (`TypeId` is unique, nothing to worry about) and passes it to your handler.
It is an interesting technique to perform what in other language ecosystems might be referred to as *dependency injection*.

3.9.4 The `INSERT` Query

We finally have a connection in `subscribe`: let's try to persist the details of our new subscriber.
We will use again the `query!` macro that we leveraged in our happy-case test.

```rust
//! src/routes/subscriptions.rs
use chrono::Utc;
use uuid::Uuid;
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    connection: web::Data<PgConnection>,
) -> HttpResponse {
    sqlx::query!(
        r#"
        INSERT INTO subscriptions (id, email, name, subscribed_at)
        VALUES ($1, $2, $3, $4)
        "#,
        Uuid::new_v4(),
        form.email,
        form.name,
        Utc::now()
    )
    // We use `get_ref` to get an immutable reference to the `PgConnection`
    // wrapped by `web::Data`.
    .execute(connection.get_ref())
    .await;
    HttpResponse::Ok().finish()
}
```

Let's unpack what is happening:

- we are binding dynamic data to our `INSERT` query. `$1` refers to the first argument passed to `query!` after the query itself, `$2` to the second and so forth. `query!` verifies at compile-time that the provided number of arguments matches what the query expects as well as that their types are compatible (e.g. you can't pass a number as `id`);
- we are generating a random `Uuid` for `id`;
- we are using the current timestamp in the `Utc` timezone for `subscribed_at`.

We have to add two new dependencies as well to our `Cargo.toml` to fix the obvious compiler errors:

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

```
uuid = { version = "0.8.1", features = ["v4"] }
chrono = "0.4.15"
```

What happens if we try to compile it again?

```
error[E0277]: the trait bound `&PgConnection: sqlx_core::executor::Executor<'_>`
              is not satisfied
  --> src/routes/subscriptions.rs:29:14
   |
29 |     .execute(connection.get_ref().deref())
   |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |              the trait `sqlx_core::executor::Executor<'_>`
   |              is not implemented for `&PgConnection`
   |
   = help: the following implementations were found:
             <&'c mut PgConnection as sqlx_core::executor::Executor<'c>>
   = note: `sqlx_core::executor::Executor<'_>` is implemented for
           `&mut PgConnection`, but not for `&PgConnection`

error: aborting due to previous error
```

`execute` wants an argument that implements `sqlx`'s `Executor` trait and it turns out, as we should have probably remembered from the query we wrote in our test, that `&PgConnection` does not implement `Executor` - only `&mut PgConnection` does.
Why is that the case?

`sqlx` has an asynchronous interface, but it does not allow you to run multiple queries *concurrently* over the same database connection.
Requiring a mutable reference allows them to enforce this guarantee in their API. You can think of a mutable reference as a *unique reference*: the compiler guarantees to `execute` that they have indeed exclusive access to that `PgConnection` because there cannot be two active mutable references to the same value at the same time in the whole program. Quite neat.

Nonetheless it might look like we designed ourselves into a corner: `web::Data` will never give us *mutable* access to the application state.
We could leverage interior mutability - e.g. putting our `PgConnection` behind a lock (e.g. a `Mutex`) would allow us to synchronise access to the underlying TCP socket and get a mutable reference to the wrapped connection once the lock has been acquired.
We could make it work, but it would not be ideal: we would be constrained to run at most one query at a time. Not great.

Let's take a second look at the documentation for `sqlx`'s `Executor` trait: what else implements `Executor` apart from `&mut PgConnection`?
Bingo: a shared reference to `PgPool`.

`PgPool` is a pool of connections to a Postgres database. How does it bypass the concurrency issue that we just discussed for `PgConnection`?
There is still interior mutability at play, but of a different kind: when you run a query against a `&PgPool`, `sqlx` will borrow a `PgConnection` from the pool and use it to execute the query; if no connection is available, it will create a new one or wait until one frees up.
This increases the number of concurrent queries that our application can run and it also improves its resiliency: a single slow query will not impact the performance of *all* incoming requests by creating contention on the connection lock.

Let's refactor `run`, `main` and `subscribe` to work with a `PgPool` instead of a single `PgConnection`:

```rust
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use sqlx::PgPool;
use std::net::TcpListener;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let configuration = get_configuration().expect("Failed to read configuration.");
    // Renamed!
    let connection_pool = PgPool::connect(&configuration.database.connection_string())
        .await
        .expect("Failed to connect to Postgres.");
    let address = format!("127.0.0.1:{}", configuration.application_port);
    let listener = TcpListener::bind(address)?;
    run(listener, connection_pool)?.await
}
```

```rust
//! src/startup.rs
use crate::routes::{health_check, subscribe};
use actix_web::dev::Server;
use actix_web::{web, App, HttpServer};
use sqlx::PgPool;
use std::net::TcpListener;

pub fn run(listener: TcpListener, db_pool: PgPool) -> Result<Server, std::io::Error> {
    // Wrap the pool using web::Data, which boils down to an Arc smart pointer
    let db_pool = web::Data::new(db_pool);
    let server = HttpServer::new(move || {
        App::new()
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            .app_data(db_pool.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

```rust
//! src/routes/subscriptions.rs
// No longer importing PgConnection!
use sqlx::PgPool;
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>, // Renamed!
) -> HttpResponse {
    sqlx::query!(/* */)
    // Using the pool as a drop-in replacement
    .execute(pool.get_ref())
    .await;
    HttpResponse::Ok().finish()
}
```

The compiler is *almost* happy: `cargo check` has a warning for us.

```
warning: unused `Result` that must be used
  --> src/routes/subscriptions.rs:13:5
   |
13 | /     sqlx::query!(
14 | |         r#"
15 | |     INSERT INTO subscriptions (id, email, name, subscribed_at)
16 | |     VALUES ($1, $2, $3, $4)
...  |
23 | |     .execute(pool.get_ref())
24 | |     .await;
   | |___________^
   |
   = note: `#[warn(unused_must_use)]` on by default
   = note: this `Result` may be an `Err` variant, which should be handled
```

`sqlx::query` may fail - it returns a `Result`, Rust's way to model fallible functions. The compiler is reminding us to handle the error case - let's follow the advice:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    // `Result` has two variants: `Ok` and `Err`.
    // The first for successes, the second for failures.
    // We use a `match` statement to choose what to do based
    // on the outcome.
    // We will talk more about `Result` going forward!
    match sqlx::query!(/* */)
    .execute(pool.get_ref())
    .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(e) => {
            println!("Failed to execute query: {}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

`cargo check` is satisfied, but the same cannot be said for `cargo test`:

```
error[E0061]: this function takes 2 arguments but 1 argument was supplied
  --> tests/health_check.rs:10:18
   |
10 |     let server = run(listener).expect("Failed to bind address");
   |                  ^^^ -------- supplied 1 argument
   |                  |
   |                  expected 2 arguments

error: aborting due to previous error
```

3.10 Updating Our Tests

The error is in our `spawn_app` helper function:

```rust
//! tests/health_check.rs
use zero2prod::startup::run;
use std::net::TcpListener;
// [...]

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 = 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 need to pass a connection pool to `run`.
Given that we are then going to need that very same connection pool in `subscribe_returns_a_200_for_valid_form_data` to perform our `SELECT` query, it makes sense to generalise `spawn_app`: instead of returning a raw `String`, we will give the caller a struct, `TestApp`. `TestApp` will hold both the address of our test application instance and a handle to the connection pool, simplifying the arrange steps in our test cases.

```rust
//! tests/health_check.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use sqlx::PgPool;
use std::net::TcpListener;

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
}

// The function is asynchronous now!
async fn spawn_app() -> TestApp {
    let listener = TcpListener::bind("127.0.0.1:0")
        .expect("Failed to bind random port");
    let port = listener.local_addr().unwrap().port();
    let address = format!("http://127.0.0.1:{}", port);

    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPool::connect(&configuration.database.connection_string())
        .await
        .expect("Failed to connect to Postgres.");

    let server = run(listener, connection_pool.clone())
        .expect("Failed to bind address");
    let _ = tokio::spawn(server);
    TestApp {
        address,
        db_pool: connection_pool,
    }
}
```

All test cases have then to be updated accordingly - an off-screen exercise that I leave to you, my dear reader.
Let's just have a look together at what `subscribe_returns_a_200_for_valid_form_data` looks like after the required changes:

```rust
//! tests/health_check.rs
// [...]
#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app = spawn_app().await;
    let client = reqwest::Client::new();

    // Act
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    let response = client
        .post(&format!("{}/subscriptions", &app.address))
        .header("Content-Type", "application/x-www-form-urlencoded")
        .body(body)
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(200, response.status().as_u16());

    let saved = sqlx::query!("SELECT email, name FROM subscriptions",)
        .fetch_one(&app.db_pool)
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
}
```

The test intent is much clearer now that we got rid of most of the boilerplate related to establishing the connection with the database.
`TestApp` is foundation we will be building on going forward to pull out supporting functionality that is useful to most of our integration tests.

The moment of truth has finally come: is our updated `subscribe` implementation enough to turn `subscribe_returns_a_200_for_valid_form_data` green?

```
running 3 tests
test health_check_works ... ok
test subscribe_returns_a_400_when_data_is_missing ... ok
test subscribe_returns_a_200_for_valid_form_data ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Yessssssss!
Success!

Let's run it again to bathe in the light of this glorious moment!

```
cargo test
```

```
running 3 tests
test health_check_works ... ok
Failed to execute query: error returned from database:
duplicate key value violates unique constraint "subscriptions_email_key"
thread 'subscribe_returns_a_200_for_valid_form_data'
       panicked at 'assertion failed: `(left == right)`
  left: `200`,
 right: `500`', tests/health_check.rs:66:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
Panic in Arbiter thread.
test subscribe_returns_a_400_when_data_is_missing ... ok
test subscribe_returns_a_200_for_valid_form_data ... FAILED

failures:

failures:
    subscribe_returns_a_200_for_valid_form_data

test result: FAILED. 2 passed; 1 failed; 0 ignored; 0 measured; 0 filtered out
```

Wait, no, what the fuck! Don't do this to us!

Ok, I lied - I knew this was going to happen.
I am sorry, I let you taste the sweet flavour of victory and then I threw you back into the mud.
There is an important lesson to be learned here, trust me.

3.10.1 Test Isolation

Your database is a gigantic global variable: all your tests are interacting with it and whatever they leave behind will be available to other tests in the suite as well as to the following test runs.
This is precisely what happened to us a moment ago: our first test run commanded our application to register a new subscriber with `ursula_le_guin@gmail.com` as their email; the application obliged.
When we re-ran our test suite we tried again to perform another `INSERT` using the same email, but our `UNIQUE` constraint on the `email` column raised a `unique key violation` and rejected the query, forcing the application to return us a `500 INTERNAL_SERVER_ERROR`.

You really do not want to have *any* kind of interaction between your tests: it makes your test runs non-deterministic and it leads down the line to spurious test failures that are extremely tricky to hunt down and fix.

There are two techniques I am aware of to ensure test isolation when interacting with a relational database in a test:

- wrap the whole test in a SQL transaction and rollback at the end of it;
- spin up a brand-new logical database for each integration test.

The first is clever and will generally be faster: rolling back a SQL transaction takes less time than spinning up a new logical database. It works quite well when writing unit tests for your queries but it is tricky to pull off in an integration test like ours: our application will borrow a `PgConnection` from a `PgPool` and we have no way to "capture" that connection in a SQL transaction context. Which leads us to the second option: potentially slower, yet much easier to implement.

How?
Before each test run, we want to:

- create a new logical database with a unique name;
- run database migrations on it.

The best place to do this is `spawn_app`, before launching our `actix-web` test application.
Let's look at it again:

```rust
//! tests/health_check.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use sqlx::PgPool;
use std::net::TcpListener;
use uuid::Uuid;

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
}

// The function is asynchronous now!
async fn spawn_app() -> TestApp {
    let listener = TcpListener::bind("127.0.0.1:0")
        .expect("Failed to bind random port");
    let port = listener.local_addr().unwrap().port();
    let address = format!("http://127.0.0.1:{}", port);

    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPool::connect(&configuration.database.connection_string())
        .await
        .expect("Failed to connect to Postgres.");

    let server = run(listener, connection_pool.clone())
        .expect("Failed to bind address");
    let _ = tokio::spawn(server);
    TestApp {
        address,
        db_pool: connection_pool,
    }
}

// [...]
```

`configuration.database.connection_string()` uses the `database_name` specified in our `configuration.yaml` file - the same for all tests.
Let's randomise it with

```rust
let mut configuration = get_configuration().expect("Failed to read configuration.");
configuration.database.database_name = Uuid::new_v4().to_string();

let connection_pool = PgPool::connect(&configuration.database.connection_string())
    .await
    .expect("Failed to connect to Postgres.");
```

`cargo test` will fail: there is no database ready to accept connections using the name we generated. Let's add a `connection_string_without_db` method to our `DatabaseSettings`:

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

impl DatabaseSettings {
    pub fn connection_string(&self) -> String {
        format!(
            "postgres://{}:{}@{}:{}/{}",
            self.username, self.password, self.host, self.port, self.database_name
        )
    }

    pub fn connection_string_without_db(&self) -> String {
        format!(
            "postgres://{}:{}@{}:{}",
            self.username, self.password, self.host, self.port
        )
    }
}
```

Omitting the database name we connect to the Postgres instance, not a specific logical database. We can now use that connection to create the database we need and run migrations on it:

```rust
//! tests/health_check.rs
// [...]
use sqlx::{Connection, Executor, PgConnection, PgPool};
use zero2prod::configuration::{get_configuration, DatabaseSettings};

async fn spawn_app() -> TestApp {
    // [...]
    let mut configuration = get_configuration().expect("Failed to read configuration.");
    configuration.database.database_name = Uuid::new_v4().to_string();
    let connection_pool = configure_database(&configuration.database).await;
    // [...]
}

pub async fn configure_database(config: &DatabaseSettings) -> PgPool {
    // Create database
    let mut connection = PgConnection::connect(&config.connection_string_without_db())
        .await
        .expect("Failed to connect to Postgres");
    connection
        .execute(format!(r#"CREATE DATABASE "{}";"#, config.database_name).as_str())
        .await
        .expect("Failed to create database.");

    // Migrate database
    let connection_pool = PgPool::connect(&config.connection_string())
        .await
        .expect("Failed to connect to Postgres.");
    sqlx::migrate!("./migrations")
        .run(&connection_pool)
        .await
        .expect("Failed to migrate the database");
    connection_pool
}
```

`sqlx::migrate!` is the same macro used by `sqlx-cli` when executing `sqlx migrate run` - no need to throw bash scripts into the mix to achieve the same result.

Let's try again to run `cargo test`:

```
running 3 tests
test subscribe_returns_a_200_for_valid_form_data ... ok
test subscribe_returns_a_400_when_data_is_missing ... ok
test health_check_works ... ok

test result: ok. 3 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

It works, this time for good.

You might have noticed that we do not perform any clean-up step at the end of our tests - the logical databases we create are not being deleted. This is intentional: we *could* add a clean-up step, but our Postgres instance is used only for test purposes and it's easy enough to restart it if, after *hundreds* of test runs, performance starts to suffer due to the number of lingering (almost empty) databases.

3.11 Summary

We covered a large number of topics in this chapter: `actix-web` extractors and HTML forms, (de)serialisation with `serde`, an overview of the available database crates in the Rust ecosystem, the fundamentals of `sqlx` as well as basic techniques to ensure test isolation when dealing with databases.
Take your time to digest the material and go back to review individual sections if necessary.

Chapter 4

Telemetry

In Chapter 3 we managed to put together a first implementation of `POST /subscriptions` to fulfill one of the user stories 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 have not yet created a web page with a HTML form to actually test the end-to-end flow, but we have a few black-box integration tests that cover the two basic scenarios we care about at this stage:

- if valid form data is submitted (i.e. both name and email have been provided), the data is saved in our database;
- if the submitted form is incomplete (e.g. the email is missing, the name is missing or both), the API returns a 400.

Should we be satisfied and rush to deploy the first version of our application on the coolest cloud provider out there?
Not yet - we are not yet equipped to properly run our software in a production environment.
We are blind: the application is not **instrumented** yet and we are not collecting any **telemetry data**, making us vulnerable to **unknown unknowns**.

If most of the previous sentence makes little to no sense to you, do not worry: getting to the bottom of it is going to be the main focus of this chapter.

4.1 Unknown Unknowns

We have a few tests. Tests are good, they make us more confident in our software, in its correctness. Nonetheless, a test suite is not *proof* of the correctness of our application. We would have to explore significantly different approaches to *prove* that something is correct (e.g. formal methods). At runtime we will encounter scenarios that we have not tested for or even thought about when designing the application in the first place.

I can point at a few blind spots based on the work we have done so far and my past experiences:

- what happens if we lose connection to the database? Does `sqlx::PgPool` try to automatically recover or will all database interactions fail from that point onwards until we restart the application?
- what happens if an attacker tries to pass malicious payloads in the body of the POST `/subscriptions` request (i.e. extremely large payloads, attempts to perform SQL injection, etc.)?

These are often referred to as **known unknowns**: shortcomings that we are aware of and we have not yet managed to investigate or we have deemed to be not relevant enough to spend time on. Given enough time and effort, we *could* get rid of most known unknowns.

Unfortunately there are issues that we have not seen before and we are not expecting, **unknown unknowns**.

Sometimes experience is enough to transform an unknown unknown into a known unknown: if you had never worked with a database before you might have not thought about what happens when we lose connection; once you have seen it happen once, it becomes a familiar failure mode to look out for.

More often than not, unknown unknowns are peculiar failure modes of the specific system we are working on.
They are problems at the crossroads between our software components, the underlying operating systems, the hardware we are using, our development process peculiarities and that huge source of randomness known as "the outside world".
They might emerge when:

- the system is pushed outside of its usual operating conditions (e.g. an unusual spike of traffic);
- multiple components experience failures at the same time (e.g. a SQL transaction is left hanging while the database is going through a master-replica failover);
- a change is introduced that moves the system equilibrium (e.g. tuning a retry policy);
- no changes have been introduced for a long time (e.g. applications have not been restarted for weeks and you start to see all sorts of memory leaks);
- etc.

All these scenarios share one key similarity: they are often impossible to reproduce outside of the live environment.
What can we do to prepare ourselves to deal with an outage or a bug caused by an unknown unknown?

4.2 Observability

We must assume that we will not be there when an unknown unknown issue arises: it might be late at night, we might be working on something else, etc.
Even if we were paying attention at the very same moment something starts to go wrong, it often isn't possible or practical to attach a debugger to a process running in production (assuming you even know in the first place *which* process you should be looking at) and the degradation might affect multiple systems at once.
The only thing we can rely on to understand and debug an unknown unknown is **telemetry data**:

information about our running applications that is collected automatically and can be later inspected to answer questions about the state of the system at a certain point in time.

What questions?
Well, if it is an unknown unknown we do not really know *in advance* what questions we might need to ask to isolate its root cause - that's the whole point.
The goal is to have an **observable application**.
Quoting from Honeycomb's observability guide

> Observability is about being able to ask arbitrary questions about your environment without — and this is the key part — having to know ahead of time what you wanted to ask.

"arbitrary" is a strong word - as all absolute statements, it might require an unreasonable investment of both time and money if we are to interpret it literally.
In practice we will also happily settle for an application that is *sufficiently* observable to enable us to deliver the level of service we promised to our users.

In a nutshell, to build an observable system we need:

- to instrument our application to collect high-quality telemetry data;
- access to tools and systems to efficiently slice, dice and manipulate the data to find answers to our questions.

We will touch upon some of the options available to fulfill the second point, but an exhaustive discussion is outside of the scope of this book.
Let's focus on the first for the rest of this chapter.

4.3 Logging

Logs are the most common type of telemetry data.
Even developers who have never heard of observability have an intuitive understanding of the usefulness of logs: logs are what you look at when stuff goes south to understand what is happening, crossing your fingers extra hard hoping you captured enough information to troubleshoot effectively.

What are logs though?
The format varies, depending on the epoch, the platform and the technologies you are using.
Nowadays a **log record** is usually a bunch of text data, with a line break to separate the current record from the next one. For example

```
The application is starting on port 8080
Handling a request to /index
Handling a request to /index
Returned a 200 OK
```

are four perfectly valid log records for a web server.

What does the Rust ecosystem have to offer us when it comes to logging?

4.3.1 The `log` Crate

The go-to crate for logging in Rust is `log`.

`log` provides five macros: `trace`, `debug`, `info`, `warn` and `error`.
They all do the same thing - emit a log a record - but each of them uses a different **log level**, as the naming implies.
trace is the lowest level: trace-level logs are often extremely verbose and have a low signal-to-noise ratio (e.g. emit a trace-level log record every time a TCP packet is received by a web server).
We then have, in increasing order of severity, *debug*, *info*, *warn* and *error*.
Error-level logs are used to report serious failures that might have user impact (e.g. we failed to handle an incoming request or a query to the database timed out).

Let's look at a quick usage example:

```rust
fn fallible_operation() -> Result<String, String> { ... }

pub fn main() {
    match fallible_operation() {
        Ok(success) => {
            log::info!("Operation succeeded: {}", success);
        }
        Err(err) => {
            log::error!("Operation failed: {}", err);
        }
    }
}
```

We are trying to perform an operation that might fail.
If it succeeds, we emit an info-level log record.
If it doesn't, we emit an error-level log record.

Notice as well how `log`'s macros support the same interpolation syntax provided by `println`/`print` in the standard library.

We can use `log`'s macros to *instrument* our codebase.
Choosing what information should be logged about the execution of a particular function is often a *local* decision: it is enough to look at the function to decide what deserves to be captured in a log record. This enables libraries to be instrumented effectively, extending the reach of our telemetry outside the boundaries of the code we have written first-hand.

4.3.2 `actix-web`'s `Logger` Middleware

`actix_web` provides a `Logger` middleware. It emits a log record for every incoming request.
Let's add it to our application.

```rust
//! src/startup.rs
use crate::routes::{health_check, subscribe};
use actix_web::dev::Server;
use actix_web::web::Data;
use actix_web::{web, App, HttpServer};
use actix_web::middleware::Logger;
use sqlx::PgPool;
use std::net::TcpListener;

```

```rust
pub fn run(listener: TcpListener, db_pool: PgPool) -> Result<Server, std::io::Error> {
    let db_pool = Data::new(db_pool);
    let server = HttpServer::new(move || {
        App::new()
            // Middlewares are added using the `wrap` method on `App`
            .wrap(Logger::default())
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            .app_data(db_pool.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

We can now launch the application using `cargo run` and fire a quick request with `curl http://127.0.0.1:8000/health_check -v`.
The request comes back with a 200 but... nothing happens on the terminal we used to launch our application.
No logs. Nothing. Blank screen.

4.3.3 The Facade Pattern

We said that instrumentation is a local decision.
There is instead a *global* decision that *applications* are uniquely positioned to do: what are we supposed to do with all those log records?
Should we append them to a file? Should we print them to the terminal? Should we send them to a remote system over HTTP (e.g. ElasticSearch)?

The `log` crate leverages the facade pattern to handle this duality.
It gives you the tools you need to emit log records, but it does not prescribe *how* those log records should be processed. It provides, instead, a `Log` trait:

```rust
//! From `log`'s source code - src/lib.rs

/// A trait encapsulating the operations required of a logger.
pub trait Log: Sync + Send {
    /// Determines if a log message with the specified metadata would be
    /// logged.
    ///
    /// This is used by the `log_enabled!` macro to allow callers to avoid
    /// expensive computation of log message arguments if the message would be
    /// discarded anyway.
    fn enabled(&self, metadata: &Metadata) -> bool;

    /// Logs the `Record`.
    ///
    /// Note that `enabled` is *not* necessarily called before this method.
    /// Implementations of `log` should perform all necessary filtering
    /// internally.
    fn log(&self, record: &Record);

    /// Flushes any buffered records.
    fn flush(&self);
}
```

At the beginning of your `main` function you can call the `set_logger` function and pass an implementation of the `Log` trait: every time a log record is emitted `Log::log` will be called on the logger you provided, therefore making it possible to perform whatever form of processing of log records you deem necessary.

If you do not call `set_logger`, then all log records will simply be discarded. Exactly what happened to our application.
Let's initialise our logger this time.

There are a few `Log` implementations available on crates.io - the most popular options are listed in the documentation of `log` itself.
We will use `env_logger` - it works nicely if, as in our case, the main goal is printing all logs records to the terminal.

Let's add it as a dependency with

```
#! Cargo.toml
# [...]
[dependencies]
env_logger = "0.9"
# [...]
```

`env_logger::Logger` prints log records to the terminal, using the following format:

```
[<timestamp> <level> <module path>] <log message>
```

It looks at the `RUST_LOG` environment variable to determine what logs should be printed and what logs should be filtered out.
`RUST_LOG=debug cargo run`, for example, will surface all logs at debug-level or higher emitted by our application or the crates we are using. `RUST_LOG=zero2prod`, instead, would filter out all records emitted by our dependencies.

Let's modify our `main.rs` file as required:

```rust
// [...]
use env_logger::Env;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // `init` does call `set_logger`, so this is all we need to do.
    // We are falling back to printing all logs at info-level or above
    // if the RUST_LOG environment variable has not been set.
    env_logger::Builder::from_env(Env::default().default_filter_or("info")).init();

    // [...]
}
```

Let's try to launch the application again using `cargo run` (equivalent to `RUST_LOG=info cargo run` given our defaulting logic). Two log records should show up on your terminal (using a new line break with indentation to make them fit within the page margins)

```
[2020-09-21T21:28:40Z INFO  actix_server::builder] Starting 12 workers
[2020-09-21T21:28:40Z INFO  actix_server::builder] Starting
    "actix-web-service-127.0.0.1:8000" service on 127.0.0.1:8000
```

If we make a request with `curl http://127.0.0.1:8000/health_check` you should see another log record, emitted by the `Logger` middleware we added a few paragraphs ago

```
[2020-09-21T21:28:43Z INFO  actix_web::middleware::logger] 127.0.0.1:47244
    "GET /health_check HTTP/1.1" 200 0 "-" "curl/7.61.0" 0.000225
```

Logs are also an awesome tool to *explore* how the software we are using works.
Try setting `RUST_LOG` to `trace` and launching the application again.
You should see a bunch of `registering with poller` log records coming from `mio`, a low-level library for non-blocking IO, as well as a couple of startup log records for each worker spawned up by `actix-web` (one for each physical core available on your machine!).
Insightful things can be learned by playing around with trace-level logs.

4.4 Instrumenting POST /subscriptions

Let's use what we learned about `log` to instrument our handler for `POST /subscriptions` requests.
It currently looks like this:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(e) => {
            // Using `println!` to capture information about the error
            // in case things don't work out as expected
            println!("Failed to execute query: {}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

Let's add `log` as a dependency:

```toml
#! Cargo.toml
# [...]
[dependencies]
log = "0.4"
# [...]
```

What should we capture in log records?

4.4.1 Interactions With External Systems

Let's start with a tried-and-tested rule of thumb: any interaction with external systems over the network should be closely monitored. We might experience networking issues, the database might be unavailable, queries might get slower over time as the `subscribers` table gets longer, etc.
Let's add two logs records: one before query execution starts and one immediately after its completion.

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    log::info!("Saving new subscriber details in the database");
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .await
    {
        Ok(_) => {
            log::info!("New subscriber details have been saved");
            HttpResponse::Ok().finish()
        },
        Err(e) => {
            println!("Failed to execute query: {}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

As it stands, we would only be emitting a log record when the query succeeds. To capture failures we need to convert that `println` statement into an error-level log:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    log::info!("Saving new subscriber details in the database");
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .await
    {
        Ok(_) => {
            log::info!("New subscriber details have been saved");
            HttpResponse::Ok().finish()
        },
        Err(e) => {
            log::error!("Failed to execute query: {:?}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

Much better - we have that query somewhat covered now.

Pay attention to a small but crucial detail: we are using `{:?}`, the `std::fmt::Debug` format, to capture the query error.
Operators are the main audience of logs - we should extract as much information as possible

about whatever malfunction occurred to ease troubleshooting. `Debug` gives us that raw view, while `std::fmt::Display({})` will return a nicer error message that is more suitable to be shown directly to our end users.

4.4.2 Think Like A User

What else should we capture?

Previously we stated that

> We will happily settle for an application that is *sufficiently* observable to enable us to deliver the level of service we promised to our users.

What does this mean *in practice*?

We need to change our reference system.
Forget, for a second, that we are the authors of this piece of software.
Put yourself in the shoes of one of your users, a person landing on your website that is interested in the content you publish and wants to subscribe to your newsletter.

What does a failure look like for them?
The story might play out like this:

> Hey!
> I tried subscribing to your newsletter using my main email address, `thomas_mann@hotmail.com`, but the website failed with a weird error. Any chance you could look into what happened?
> Best,
> Tom
> P.S. Keep it up, your blog rocks!

Tom landed on our website and received "a weird error" when he pressed the `Submit` button.

Our application is *sufficiently observable* if we can triage the issue from the breadcrumbs of information he has provided us - i.e. the email address he entered.

Can we do it?

Let's, first of all, confirm the issue: is Tom registered as a subscriber?
We can connect to the database and run a quick query to double-check that there is no record with `thomas_mann@hotmail.com` as `email` in our `subscribers` table.
The issue is confirmed. What now?

None of our logs include the subscriber email address, so we cannot search for it. Dead end.
We could ask Tom to provide additional information: all our log records have a timestamp, maybe if he remembers around what time he tried to subscribe we can dig something out?

This is a clear indication that our current logs are not good enough.
Let's improve them:

```rust
//! src/routes/subscriptions.rs
//! ..

pub async fn subscribe(/* */) -> HttpResponse {
    // We are using the same interpolation syntax of `println`/`print` here!
    log::info!(
        "Adding '{}' '{}' as a new subscriber.",
        form.email,
        form.name
    );
    log::info!("Saving new subscriber details in the database");
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .await
    {
        Ok(_) => {
            log::info!("New subscriber details have been saved");
            HttpResponse::Ok().finish()
        },
        Err(e) => {
            log::error!("Failed to execute query: {:?}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

Much better - we now have a log line that is capturing both name and email.[1].
Is it enough to troubleshoot Tom's issue?

4.4.3 Logs Must Be Easy To Correlate

> Going forward I will omit logs emitted by `sqlx` from the reported terminal output to keep the examples concise. `sqlx`'s logs use the `INFO` level by default - we will tune it down to `TRACE` in Chapter 5.

If we had a single copy of our web server running at any point in time and that copy was only capable of handling a single request at a time, we might imagine logs showing up in our terminal more or less like this:

```
# First request
[.. INFO zero2prod] Adding 'thomas_mann@hotmail.com' 'Tom' as a new subscriber
[.. INFO zero2prod] Saving new subscriber details in the database
[.. INFO zero2prod] New subscriber details have been saved
[.. INFO actix_web] .. "POST /subscriptions HTTP/1.1" 200 ..
# Second request
[.. INFO zero2prod] Adding 's_erikson@malazan.io' 'Steven' as a new subscriber
```

[1] Should we log names and emails? If you are operating in Europe, they generally qualify as **P**ersonal **I**dentifiable **I**nformation (PII) and their processing must obey the principles and rules laid out in the **G**eneral **D**ata **P**rotection **R**egulation (GDPR). We should have tight controls around who can access that information, how long we are planning to store it for, procedures to delete it if the user asks to be forgotten, etc. Generally speaking, there are many types of information that would be useful for debugging purposes but cannot be logged freely (e.g. passwords) - you will either have to do without them or rely on obfuscation (e.g. tokenization/pseudonymisation) to strike a balance between security, privacy and usefulness.

```
[.. ERROR zero2prod] Failed to execute query: connection error with the database
[.. ERROR actix_web] .. "POST /subscriptions HTTP/1.1" 500 ..
```

You can clearly see where a single request begins, what happened while we tried to fulfill it, what we returned as a response, where the next request begins, etc.
It is easy to follow.
But this is not what it looks like when you are handling multiple requests concurrently:

```
[.. INFO zero2prod] Receiving request for POST /subscriptions
[.. INFO zero2prod] Receiving request for POST /subscriptions
[.. INFO zero2prod] Adding 'thomas_mann@hotmail.com' 'Tom' as a new subscriber
[.. INFO zero2prod] Adding 's_erikson@malazan.io' 'Steven' as a new subscriber
[.. INFO zero2prod] Saving new subscriber details in the database
[.. ERROR zero2prod] Failed to execute query: connection error with the database
[.. ERROR actix_web] .. "POST /subscriptions HTTP/1.1" 500 ..
[.. INFO zero2prod] Saving new subscriber details in the database
[.. INFO zero2prod] New subscriber details have been saved
[.. INFO actix_web] .. "POST /subscriptions HTTP/1.1" 200 ..
```

What details did we fail to save though? `thomas_mann@hotmail.com` or `s_erikson@malazan.io`? Impossible to say from the logs.

We need a way to *correlate* all logs related to the same request.
This is usually achieved using a **request id** (also known as **correlation id**): when we start to process an incoming request we generate a random identifier (e.g. a UUID) which is then associated to all logs concerning the fulfilling of that specific request.

Let's add one to our handler:

```rust
//! src/routes/subscriptions.rs
//! ..

pub async fn subscribe(/* */) -> HttpResponse {
    // Let's generate a random unique identifier
    let request_id = Uuid::new_v4();
    log::info!(
        "request_id {} - Adding '{}' '{}' as a new subscriber.",
        request_id,
        form.email,
        form.name
    );
    log::info!(
        "request_id {} - Saving new subscriber details in the database",
        request_id
    );
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .await
    {
        Ok(_) => {
            log::info!(
                "request_id {} - New subscriber details have been saved",
                request_id
            );
            HttpResponse::Ok().finish()
        },
        Err(e) => {
            log::error!(
                "request_id {} - Failed to execute query: {:?}",
                request_id,
                e
            );
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

Logs for an incoming request will now look like this:

```
curl -i -X POST -d 'email=thomas_mann@hotmail.com&name=Tom' \
    http://127.0.0.1:8000/subscriptions
```

```
[.. INFO  zero2prod] request_id 9ebde7e9-1efe-40b9-ab65-86ab422e6b87 - Adding
    'thomas_mann@hotmail.com' 'Tom' as a new subscriber.
[.. INFO  zero2prod] request_id 9ebde7e9-1efe-40b9-ab65-86ab422e6b87 - Saving
    new subscriber details in the database
[.. INFO  zero2prod] request_id 9ebde7e9-1efe-40b9-ab65-86ab422e6b87 - New
    subscriber details have been saved
[.. INFO  actix_web] .. "POST /subscriptions HTTP/1.1" 200 ..
```

We can now search for `thomas_mann@hotmail.com` in our logs, find the first record, grab the `request_id` and then pull down all the other log records associated with that request.
Well, *almost* all the logs: `request_id` is created in our `subscribe` handler, therefore `actix_web`'s `Logger` middleware is completely unaware of it.
That means that we will not know what status code our application has returned to the user when they tried to subscribe to our newsletter.

What should we do?
We could bite the bullet, remove `actix_web`'s `Logger`, write a middleware to generate a random request identifier for every incoming request and then write our own logging middleware that is aware of the identifier and includes it in all log lines.

Could it work? Yes.
Should we do it? Probably not.

4.5 Structured Logging

To ensure that `request_id` is included in all log records we would have to:

- rewrite all upstream components in the request processing pipeline (e.g. `actix-web`'s `Logger`);
- change the signature of all downstream functions we are calling from the `subscribe` handler; if they are emitting a log statement, they need to include the `request_id`, which therefore needs to be passed down as an argument.

What about log records emitted by the crates we are importing into our project? Should we rewrite those as well?
It is clear that **this approach cannot scale**.

Let's take a step back: what does our code look like?
We have an over-arching task (an HTTP request), which is broken down in a set of sub-tasks (e.g. parse input, make a query, etc.), which might in turn be broken down in smaller sub-routines recursively.
Each of those units of work has a *duration* (i.e. a beginning and an end).
Each of those units of work has a *context* associated to it (e.g. name and email of a new subscriber, `request_id`) that is naturally shared by all its sub-units of work.

No doubt we are struggling: log statements are isolated events happening at a defined moment in time that we are stubbornly trying to use to represent a tree-like processing pipeline.
Logs are the wrong abstraction.

What should we use then?

4.5.1 The `tracing` Crate

The `tracing` crate comes to the rescue:

> `tracing` expands upon logging-style diagnostics by allowing libraries and applications to record structured events with additional information about temporality and causality — unlike a log message, a span in tracing has a beginning and end time, may be entered and exited by the flow of execution, and may exist within a nested tree of similar spans.

That is music to our ears.
What does it look like in practice?

4.5.2 Migrating From `log` To `tracing`

There is only one way to find out - let's convert our `subscribe` handler to use `tracing` instead of `log` for instrumentation. Let's add `tracing` to our dependencies:

```
#! Cargo.toml

[dependencies]
tracing = { version = "0.1", features = ["log"] }
# [...]
```

The first migration step is as straight-forward as it gets: search and replace all occurrences of the `log` string in our function body with `tracing`.

```
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    let request_id = Uuid::new_v4();
    tracing::info!(
        "request_id {} - Adding '{}' '{}' as a new subscriber.",
        request_id,
        form.email,
        form.name
    );
    tracing::info!(
```

```rust
        "request_id {} - Saving new subscriber details in the database",
        request_id
    );
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .await
    {
        Ok(_) => {
            tracing::info!(
                "request_id {} - New subscriber details have been saved",
                request_id
            );
            HttpResponse::Ok().finish()
        },
        Err(e) => {
            tracing::error!(
                "request_id {} - Failed to execute query: {:?}",
                request_id,
                e
            );
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

That's it.
If you run the application and fire a `POST /subscriptions` request you will see *exactly the same logs* in your console. Identical.
Pretty cool, isn't it?

This works thanks to `tracing`'s `log` feature flag, which we enabled in `Cargo.toml`. It ensures that every time an event or a span are created using `tracing`'s macros a corresponding log event is emitted, allowing `log`'s loggers to pick up on it (`env_logger`, in our case).

4.5.3 `tracing`'s `Span`

We can now start to leverage `tracing`'s `Span` to better capture the structure of our program.
We want to create a span that represents the whole HTTP request:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    let request_id = Uuid::new_v4();
    // Spans, like logs, have an associated level
    // `info_span` creates a span at the info-level
    let request_span = tracing::info_span!(
        "Adding a new subscriber.",
        %request_id,
        subscriber_email = %form.email,
        subscriber_name= %form.name
    );
    // Using `enter` in an async function is a recipe for disaster!
    // Bear with me for now, but don't do this at home.
    // See the following section on `Instrumenting Futures`
    let _request_span_guard = request_span.enter();

    // [ ... ]
    // `_request_span_guard` is dropped at the end of `subscribe`
    // That's when we "exit" the span
}
```

There is a lot going on here - let's break it down.

We are using the `info_span!` macro to create a new span and attach some values to its context: `request_id`, `form.email` and `form.name`.
We are not using string interpolation anymore: `tracing` allows us to associate *structured* information to our spans as a collection of key-value pairs[2]. We can explicitly name them (e.g. `subscriber_email` for `form.email`) or implicitly use the variable name as key (e.g. the isolated `request_id` is equivalent to `request_id = request_id`).
Notice that we prefixed all of them with a `%` symbol: we are telling `tracing` to use their `Display` implementation for logging purposes. You can find more details on the other available options in their documentation.

`info_span` returns the newly created span, but we have to *explicit* step into it using the `.enter()` method to activate it.

`.enter()` returns an instance of `Entered`, a *guard*: as long the guard variable is not dropped all downstream spans and log events will be registered as *children* of the entered span. This is a typical Rust pattern, often referred to as **R**esource **A**cquisition **I**s **I**nitialization (RAII): the compiler keeps track of the lifetime of all variables and when they go out of scope it inserts a call to their destructor, `Drop::drop`.
The default implementation of the `Drop` trait simply takes care of releasing the resources owned by that variable. We can, though, specify a custom `Drop` implementation to perform other cleanup operations on drop - e.g. exiting from a span when the `Entered` guard gets dropped:

```
//! `tracing`'s source code

impl<'a> Drop for Entered<'a> {
    #[inline]
    fn drop(&mut self) {
        // Dropping the guard exits the span.
        //
        // Running this behaviour on drop rather than with an explicit function
        // call means that spans may still be exited when unwinding.
        if let Some(inner) = self.span.inner.as_ref() {
            inner.subscriber.exit(&inner.id);
        }

        if_log_enabled! {{
            if let Some(ref meta) = self.span.meta {
                self.span.log(
                    ACTIVITY_LOG_TARGET,
                    log::Level::Trace,
                    format_args!("<- {}", meta.name())
```

[2] The capability of capturing contextual information as a collection of key-value pairs has recently been explored in the `log` crate as well - see the unstable `kv` feature. At the time of writing though, none of the mainstream `Log` implementation supports structured logging as far as I can see.

```
        );
      }
    }}
  }
}
```

Inspecting the source code of your dependencies can often expose some gold nuggets - we just found out that if the `log` feature flag is enabled `tracing` will emit a trace-level log when a span exits.
Let's give it a go immediately:

```
RUST_LOG=trace cargo run
```

```
[.. INFO  zero2prod] Adding a new subscriber.; request_id=f349b0fe..
  subscriber_email=ursulale_guin@gmail.com subscriber_name=le guin
[.. TRACE zero2prod] -> Adding a new subscriber.
[.. INFO  zero2prod] request_id f349b0fe.. - Saving new subscriber details
  in the database
[.. INFO  zero2prod] request_id f349b0fe.. - New subscriber details have
  been saved
[.. TRACE zero2prod] <- Adding a new subscriber.
[.. TRACE zero2prod] -- Adding a new subscriber.
[.. INFO  actix_web] .. "POST /subscriptions HTTP/1.1" 200 ..
```

Notice how all the information we captured in the span's context is reported in the emitted log line.
We can closely follow the lifetime of our span using the emitted logs:

- `Adding a new subscriber` is logged when the span is created;
- We enter the span (→);
- We execute the INSERT query;
- We exit the span (←);
- We finally close the span (`--`).

Wait, what is the difference between exiting and closing a span?
Glad you asked!

You can enter (and exit) a span multiple times. Closing, instead, is final: it happens when the span itself is dropped.
This comes pretty handy when you have a unit of work that can be paused and then resumed - e.g. an asynchronous task!

4.5.4 Instrumenting Futures

Let's use our database query as an example.
The executor might have to poll its future more than once to drive it to completion - while that future is idle, we are going to make progress on other futures.

This can clearly cause issues: how do we make sure we don't mix their respective spans?
The best way would be to closely mimic the future's lifecycle: we should enter into the span associated to our future every time it is polled by the executor and exit every time it gets parked.

That's where `Instrument` comes into the picture. It is an extension trait for futures. `Instrument::instrument` does exactly what we want: enters the span we pass as argument every time `self`, the future, is polled; it exits the span every time the future is parked.

Let's try it out on our query:

```
//! src/routes/subscriptions.rs
use tracing::Instrument;
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    let request_id = Uuid::new_v4();
    let request_span = tracing::info_span!(
        "Adding a new subscriber.",
        %request_id,
        subscriber_email = %form.email,
        subscriber_name= %form.name
    );
    let _request_span_guard = request_span.enter();

    // We do not call `.enter` on query_span!
    // `.instrument` takes care of it at the right moments
    // in the query future lifetime
    let query_span = tracing::info_span!(
        "Saving new subscriber details in the database"
    );
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        // First we attach the instrumentation, then we `.await` it
        .instrument(query_span)
        .await
    {
        Ok(_) => {
            HttpResponse::Ok().finish()
        },
        Err(e) => {
            // Yes, this error log falls outside of `query_span`
            // We'll rectify it later, pinky swear!
            tracing::error!("Failed to execute query: {:?}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

If we launch the application again with `RUST_LOG=trace` and try a `POST /subscriptions` request we will see logs that look somewhat similar to these:

```
[.. INFO  zero2prod] Adding a new subscriber.; request_id=f349b0fe..
    subscriber_email=ursulale_guin@gmail.com subscriber_name=le guin
[.. TRACE zero2prod] -> Adding a new subscriber.
[.. INFO  zero2prod] Saving new subscriber details in the database
[.. TRACE zero2prod] -> Saving new subscriber details in the database
[.. TRACE zero2prod] <- Saving new subscriber details in the database
[.. TRACE zero2prod] -> Saving new subscriber details in the database
[.. TRACE zero2prod] <- Saving new subscriber details in the database
[.. TRACE zero2prod] -> Saving new subscriber details in the database
[.. TRACE zero2prod] <- Saving new subscriber details in the database
[.. TRACE zero2prod] -> Saving new subscriber details in the database
[.. TRACE zero2prod] -> Saving new subscriber details in the database
[.. TRACE zero2prod] <- Saving new subscriber details in the database
[.. TRACE zero2prod] -- Saving new subscriber details in the database
[.. TRACE zero2prod] <- Adding a new subscriber.
[.. TRACE zero2prod] -- Adding a new subscriber.
[.. INFO  actix_web] .. "POST /subscriptions HTTP/1.1" 200 ..
```

We can clearly see how many times the query future has been polled by the executor before completing. How cool is that!?

4.5.5 `tracing`'s `Subscriber`

We embarked in this migration from `log` to `tracing` because we needed a better abstraction to instrument our code effectively. We wanted, in particular, to attach `request_id` to all logs associated to the same incoming HTTP request.
Although I promised `tracing` was going to solve our problem, look at those logs: `request_id` is only printed on the very first log statement where we attach it explicitly to the span context.
Why is that?

Well, we haven't completed our migration *yet*.
Although we moved all our instrumentation code from `log` to `tracing` we are still using `env_logger` to process everything!

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    env_logger::from_env(Env::default().default_filter_or("info")).init();
    // [...]
}
```

`env_logger`'s logger implements `log`'s `Log` trait - it knows nothing about the rich structure exposed by `tracing`'s `Span`!
`tracing`'s compatibility with `log` was great to get off the ground, but it is now time to replace `env_logger` with a `tracing`-native solution.

The `tracing` crate follows the same facade pattern used by `log` - you can freely use its macros to instrument your code, but applications are in charge to spell out how that span telemetry data should be processed.
`Subscriber` is the `tracing` counterpart of `log`'s `Log`: an implementation of the `Subscriber` trait exposes a variety of methods to manage every stage of the lifecycle of a `Span` - creation, enter/exit, closure, etc.

```rust
//! `tracing`'s source code

pub trait Subscriber: 'static {
    fn new_span(&self, span: &span::Attributes<'_>) -> span::Id;
    fn event(&self, event: &Event<'_>);
    fn enter(&self, span: &span::Id);
    fn exit(&self, span: &span::Id);
    fn clone_span(&self, id: &span::Id) -> span::Id;
    // [...]
}
```

The quality of `tracing`'s documentation is breath-taking - I *strongly* invite you to have a look for yourself at `Subscriber`'s docs to properly understand what each of those methods does.

4.5.6 `tracing-subscriber`

`tracing` does not provide any subscriber out of the box.
We need to look into `tracing-subscriber`, another crate maintained in-tree by the `tracing` project, to find a few basic subscribers to get off the ground. Let's add it to our dependencies:

```
[dependencies]
# ...
tracing-subscriber = { version = "0.3", features = ["registry", "env-filter"] }
```

`tracing-subscriber` does much more than providing us with a few handy subscribers.
It introduces another key trait into the picture, `Layer`.
`Layer` makes it possible to build a *processing pipeline* for spans data: we are not forced to provide an all-encompassing subscriber that does everything we want; we can instead combine multiple smaller layers to obtain the processing pipeline we need.
This substantially reduces duplication across in `tracing` ecosystem: people are focused on adding new capabilities by churning out new layers rather than trying to build the best-possible-batteries-included subscriber.

The cornerstone of the layering approach is `Registry`.
`Registry` implements the `Subscriber` trait and takes care of all the difficult stuff:

> `Registry` does not actually record traces itself: instead, it collects and stores span data that is exposed to any layer wrapping it [...]. The `Registry` is responsible for storing span metadata, recording relationships between spans, and tracking which spans are active and which are closed.

Downstream layers can piggyback on `Registry`'s functionality and focus on their purpose: filtering what spans should be processed, formatting span data, shipping span data to remote systems, etc.

4.5.7 `tracing-bunyan-formatter`

We'd like to put together a subscriber that has feature-parity with the good old `env_logger`.
We will get there by combining three layers[3]:

- `tracing_subscriber::filter::EnvFilter` discards spans based on their log levels and their origins, just as we did in `env_logger` via the `RUST_LOG` environment variable;
- `tracing_bunyan_formatter::JsonStorageLayer` processes spans data and stores the associated metadata in an easy-to-consume JSON format for downstream layers. It does, in particular, propagate context from parent spans to their children;
- `tracing_bunyan_formatter::BunyanFormatterLayer` builds on top of `JsonStorageLayer` and outputs log records in `bunyan`-compatible JSON format.

Let's add `tracing_bunyan_formatter` to our dependencies[4]:

[3] We are using `tracing-bunyan-formatter` instead of the formatting layer provided by `tracing-subscriber` because the latter does not implement metadata inheritance: it would therefore fail to meet our requirements.

[4] Full disclosure - I am the author of `tracing-bunyan-formatter`.

```
[dependencies]
# ...
tracing-bunyan-formatter = "0.3"
```

We can now tie everything together in our `main` function:

```rust
//! src/main.rs
//! [...]
use tracing::subscriber::set_global_default;
use tracing_bunyan_formatter::{BunyanFormattingLayer, JsonStorageLayer};
use tracing_subscriber::{layer::SubscriberExt, EnvFilter, Registry};

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // We removed the `env_logger` line we had before!

    // We are falling back to printing all spans at info-level or above
    // if the RUST_LOG environment variable has not been set.
    let env_filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new("info"));
    let formatting_layer = BunyanFormattingLayer::new(
        "zero2prod".into(),
        // Output the formatted spans to stdout.
        std::io::stdout
    );
    // The `with` method is provided by `SubscriberExt`, an extension
    // trait for `Subscriber` exposed by `tracing_subscriber`
    let subscriber = Registry::default()
        .with(env_filter)
        .with(JsonStorageLayer)
        .with(formatting_layer);
    // `set_global_default` can be used by applications to specify
    // what subscriber should be used to process spans.
    set_global_default(subscriber).expect("Failed to set subscriber");

    // [...]
}
```

If you launch the application with `cargo run` and fire a request you'll see these logs (pretty-printed here to be easier on the eye):

```
{
  "msg": "[ADDING A NEW SUBSCRIBER - START]",
  "subscriber_name": "le guin",
  "request_id": "30f8cce1-f587-4104-92f2-5448e1cc21f6",
  "subscriber_email": "ursula_le_guin@gmail.com"
  ...
}
{
  "msg": "[SAVING NEW SUBSCRIBER DETAILS IN THE DATABASE - START]",
  "subscriber_name": "le guin",
  "request_id": "30f8cce1-f587-4104-92f2-5448e1cc21f6",
  "subscriber_email": "ursula_le_guin@gmail.com"
  ...
}
{
  "msg": "[SAVING NEW SUBSCRIBER DETAILS IN THE DATABASE - END]",
  "elapsed_milliseconds": 4,
  "subscriber_name": "le guin",
  "request_id": "30f8cce1-f587-4104-92f2-5448e1cc21f6",
  "subscriber_email": "ursula_le_guin@gmail.com"
  ...
}
{
  "msg": "[ADDING A NEW SUBSCRIBER - END]",
  "elapsed_milliseconds": 5
  "subscriber_name": "le guin",
  "request_id": "30f8cce1-f587-4104-92f2-5448e1cc21f6",
  "subscriber_email": "ursula_le_guin@gmail.com",
  ...
}
```

We made it: everything we attached to the original context has been propagated to all its sub-spans.
`tracing-bunyan-formatter` also provides duration out-of-the-box: every time a span is closed a JSON message is printed to the console with an `elapsed_millisecond` property attached to it.
The JSON format is extremely friendly when it comes to searching: an engine like ElasticSearch can easily ingest all these records, infer a schema and index the `request_id`, `name` and `email` fields. It unlocks the full power of a querying engine to sift through our logs!
This is exponentially better than we had before: to perform complex searches we would have had to use custom-built regexes, therefore limiting considerably the range of questions that we could easily ask to our logs.

4.5.8 `tracing-log`

If you take a closer look you will realise we lost something along the way: our terminal is only showing logs that were directly emitted by our application. What happened to `actix-web`'s log records?

`tracing`'s `log` feature flag ensures that a log record is emitted every time a `tracing` event happens, allowing `log`'s loggers to pick them up.
The opposite does not hold true: `log` does not emit `tracing` events out of the box and does not provide a feature flag to enable this behaviour.

If we want it, we need to explicitly register a logger implementation to redirect logs to our `tracing` subscriber for processing.

We can use `LogTracer`, provided by the `tracing-log` crate.

```
#! Cargo.toml
# [...]
[dependencies]
tracing-log = "0.1"
# [...]
```

Let's edit our `main` as required:

```
//! src/main.rs
//! [...]
use tracing::subscriber::set_global_default;
use tracing_bunyan_formatter::{BunyanFormattingLayer, JsonStorageLayer};
```

```rust
use tracing_subscriber::{layer::SubscriberExt, EnvFilter, Registry};
use tracing_log::LogTracer;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // Redirect all `log`'s events to our subscriber
    LogTracer::init().expect("Failed to set logger");

    let env_filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new("info"));
    let formatting_layer = BunyanFormattingLayer::new(
        "zero2prod".into(),
        std::io::stdout
    );
    let subscriber = Registry::default()
        .with(env_filter)
        .with(JsonStorageLayer)
        .with(formatting_layer);
    set_global_default(subscriber).expect("Failed to set subscriber");

    // [...]
}
```

All `actix-web`'s logs should once again be available in our console.

4.5.9 Removing Unused Dependencies

If you quickly scan through all our files you will realise that we are not using `log` or `env_logger` anywhere at this point. We should remove them from our `Cargo.toml` file.

In a large project it is very difficult to spot that a dependency has become unused after a refactoring.
Luckily enough, tooling comes to the rescue once again - let's install `cargo-udeps` (**u**nused **dep**endencies):

```
cargo install cargo-udeps
```

`cargo-udeps` scans your `Cargo.toml` file and checks if all the crates listed under `[dependencies]` have actually been used in the project. Check `cargo-deps`' trophy case for a long list of popular Rust projects where `cargo-udeps` was able to spot unused dependencies and cut down build times.

Let's run it on our project!

```
# cargo-udeps requires the nightly compiler.
# We add +nightly to our cargo invocation
# to tell cargo explicitly what toolchain we want to use.
cargo +nightly udeps
```

The output should be

```
zero2prod
  dependencies
    "env-logger"
```

Unfortunately it does not pick up `log`.

Let's strike both out of our `Cargo.toml` file.

4.5.10 Cleaning Up Initialisation

We relentlessly pushed forward to improve the observability posture of our application.
Let's now take a step back and look at the code we wrote to see if we can improve in any meaningful way.

Let's start from our `main` function:

```rust
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use sqlx::postgres::PgPool;
use std::net::TcpListener;
use tracing::subscriber::set_global_default;
use tracing_bunyan_formatter::{BunyanFormattingLayer, JsonStorageLayer};
use tracing_log::LogTracer;
use tracing_subscriber::{layer::SubscriberExt, EnvFilter, Registry};

#[tokio::main]
async fn main() -> std::io::Result<()> {
    LogTracer::init().expect("Failed to set logger");

    let env_filter = EnvFilter::try_from_default_env()
        .unwrap_or(EnvFilter::new("info"));
    let formatting_layer = BunyanFormattingLayer::new(
        "zero2prod".into(),
        std::io::stdout
    );
    let subscriber = Registry::default()
        .with(env_filter)
        .with(JsonStorageLayer)
        .with(formatting_layer);
    set_global_default(subscriber).expect("Failed to set subscriber");

    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPool::connect(&configuration.database.connection_string())
        .await
        .expect("Failed to connect to Postgres.");

    let address = format!("127.0.0.1:{}", configuration.application_port);
    let listener = TcpListener::bind(address)?;
    run(listener, connection_pool)?.await?;
    Ok(())
}
```

There is a lot going on in that `main` function right now.
Let's break it down a bit:

```rust
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use sqlx::postgres::PgPool;
use std::net::TcpListener;
use tracing::{Subscriber, subscriber::set_global_default};
use tracing_bunyan_formatter::{BunyanFormattingLayer, JsonStorageLayer};
use tracing_log::LogTracer;
use tracing_subscriber::{layer::SubscriberExt, EnvFilter, Registry};

/// Compose multiple layers into a `tracing`'s subscriber.
///
/// # Implementation Notes
///
/// We are using `impl Subscriber` as return type to avoid having to
/// spell out the actual type of the returned subscriber, which is
/// indeed quite complex.
/// We need to explicitly call out that the returned subscriber is
/// `Send` and `Sync` to make it possible to pass it to `init_subscriber`
/// later on.
pub fn get_subscriber(
    name: String,
    env_filter: String
) -> impl Subscriber + Send + Sync {
    let env_filter = EnvFilter::try_from_default_env()
        .unwrap_or_else(|_| EnvFilter::new(env_filter));
    let formatting_layer = BunyanFormattingLayer::new(
        name,
        std::io::stdout
    );
    Registry::default()
        .with(env_filter)
        .with(JsonStorageLayer)
        .with(formatting_layer)
}

/// Register a subscriber as global default to process span data.
///
/// It should only be called once!
pub fn init_subscriber(subscriber: impl Subscriber + Send + Sync) {
    LogTracer::init().expect("Failed to set logger");
    set_global_default(subscriber).expect("Failed to set subscriber");
}

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into());
    init_subscriber(subscriber);

    // [...]
}
```

We can now move `get_subscriber` and `init_subscriber` to a module within our `zero2prod` library, `telemetry`.

```rust
//! src/lib.rs
pub mod configuration;
pub mod routes;
pub mod startup;
pub mod telemetry;
```

```rust
//! src/telemetry.rs
use tracing::subscriber::set_global_default;
use tracing::Subscriber;
use tracing_bunyan_formatter::{BunyanFormattingLayer, JsonStorageLayer};
use tracing_log::LogTracer;
use tracing_subscriber::{layer::SubscriberExt, EnvFilter, Registry};

pub fn get_subscriber(
    name: String,
    env_filter: String
) -> impl Subscriber + Sync + Send {
    // [...]
}

pub fn init_subscriber(subscriber: impl Subscriber + Sync + Send) {
    // [...]
}
```

```rust
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use zero2prod::telemetry::{get_subscriber, init_subscriber};
use sqlx::postgres::PgPool;
use std::net::TcpListener;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into());
    init_subscriber(subscriber);

    // [...]
}
```

Awesome.

4.5.11 Logs For Integration Tests

We are not just cleaning up for aesthetic/readability reasons - we are moving those two functions to the `zero2prod` library to make them available to our test suite!

As a rule of thumb, everything we use in our application should be reflected in our integration tests.
Structured logging, in particular, can significantly speed up our debugging when an integration test fails: we might not have to attach a debugger, more often than not the logs can tell us where something went wrong. It is also a good benchmark: if you cannot debug it from logs, imagine how difficult would it be to debug in production!

Let's change our `spawn_app` helper function to take care of initialising our `tracing` stack:

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

use zero2prod::configuration::{get_configuration, DatabaseSettings};
use zero2prod::startup::run;
use zero2prod::telemetry::{get_subscriber, init_subscriber};
use sqlx::{Connection, Executor, PgConnection, PgPool};
use std::net::TcpListener;
use uuid::Uuid;

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
}

async fn spawn_app() -> TestApp {
    let subscriber = get_subscriber("test".into(), "debug".into());
    init_subscriber(subscriber);

    let listener = TcpListener::bind("127.0.0.1:0").expect("Failed to bind random port");
    let port = listener.local_addr().unwrap().port();
    let address = format!("http://127.0.0.1:{}", port);

    let mut configuration = get_configuration().expect("Failed to read configuration.");
    configuration.database.database_name = Uuid::new_v4().to_string();
    let connection_pool = configure_database(&configuration.database).await;

    let server = run(listener, connection_pool.clone()).expect("Failed to bind address");
    let _ = tokio::spawn(server);
    TestApp {
        address,
        db_pool: connection_pool,
    }
}

// [...]
```

If you try to run `cargo test` you will be greeted by *one* success and a long series of test failures:

```
failures:
---- subscribe_returns_a_400_when_data_is_missing stdout ----
thread 'subscribe_returns_a_400_when_data_is_missing' panicked at
'Failed to set logger: SetLoggerError(())'
Panic in Arbiter thread.

---- subscribe_returns_a_200_for_valid_form_data stdout ----
thread 'subscribe_returns_a_200_for_valid_form_data' panicked at
'Failed to set logger: SetLoggerError(())'
Panic in Arbiter thread.


failures:
    subscribe_returns_a_200_for_valid_form_data
    subscribe_returns_a_400_when_data_is_missing
```

`init_subscriber` should only be called once, but it is being invoked by all our tests.
We can use `once_cell` to rectify it[5]:

```toml
#! Cargo.toml
# [...]
[dev-dependencies]
once_cell = "1"
```

[5] Given that we never refer to `TRACING` after its initialization, we could have used `std::sync::Once` with its `call_once` method. Unfortunately, as soon as the requirements change (i.e. you need to use it after initialization), you end up reaching for `std::sync::SyncOnceCell`, which is not stable yet. `once_cell` covers both usecases - this seemed like a great opportunity to introduce a useful crate into your toolkit.

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

```rust
//! tests/health_check.rs
// [...]
use once_cell::sync::Lazy;

// Ensure that the `tracing` stack is only initialised once using `once_cell`
static TRACING: Lazy<()> = Lazy::new(|| {
    let subscriber = get_subscriber("test".into(), "debug".into());
    init_subscriber(subscriber);
});

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
}

async fn spawn_app() -> TestApp {
    // The first time `initialize` is invoked the code in `TRACING` is executed.
    // All other invocations will instead skip execution.
    Lazy::force(&TRACING);

    // [...]
}

// [...]
```

`cargo test` is green again.
The output, though, is very noisy: we have several log lines coming out of each test case.
We want our tracing instrumentation to be exercised in every test, but we do not want to look at those logs *every time* we run our test suite.

`cargo test` solves the very same problem for `println`/`print` statements. By default, it swallows everything that is printed to console. You can explicitly opt in to look at those print statements using `cargo test -- --nocapture`.

We need an equivalent strategy for our `tracing` instrumentation.
Let's add a new parameter to `get_subscriber` to allow customisation of what sink logs should be written to:

```rust
//! src/telemetry.rs
use tracing_subscriber::fmt::MakeWriter;
// [...]

pub fn get_subscriber<Sink>(
    name: String,
    env_filter: String,
    sink: Sink,
) -> impl Subscriber + Sync + Send
    where
        // This "weird" syntax is a higher-ranked trait bound (HRTB)
        // It basically means that Sink implements the `MakeWriter`
        // trait for all choices of the lifetime parameter `'a`
        // Check out https://doc.rust-lang.org/nomicon/hrtb.html
        // for more details.
        Sink: for<'a> MakeWriter<'a> + Send + Sync + 'static,
{
    // [...]
    let formatting_layer = BunyanFormattingLayer::new(name, sink);
    // [...]
}
```

We can then adjust our `main` function to use `stdout`:

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into(), std::io::stdout);

    // [...]
}
```

In our test suite we will choose the sink dynamically according to an environment variable, `TEST_LOG`. If `TEST_LOG` is set, we use `std::io::stdout`.
If `TEST_LOG` is not set, we send all logs into the void using `std::io::sink`.
Our own home-made version of the `--nocapture` flag.

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

// Ensure that the `tracing` stack is only initialised once using `once_cell`
static TRACING: Lazy<()> = Lazy::new(|| {
    let default_filter_level = "info".to_string();
    let subscriber_name = "test".to_string();
    // We cannot assign the output of `get_subscriber` to a variable based on the value
    // of `TEST_LOG` because the sink is part of the type returned by `get_subscriber`,
    // therefore they are not the same type. We could work around it, but this is the
    // most straight-forward way of moving forward.
    if std::env::var("TEST_LOG").is_ok() {
        let subscriber = get_subscriber(subscriber_name, default_filter_level, std::io::stdout);
        init_subscriber(subscriber);
    } else {
        let subscriber = get_subscriber(subscriber_name, default_filter_level, std::io::sink);
        init_subscriber(subscriber);
    };
});

// [...]
```

When you want to see all logs coming out of a certain test case to debug it you can run

```bash
# We are using the `bunyan` CLI to prettify the outputted logs
# The original `bunyan` requires NPM, but you can install a Rust-port with
# `cargo install bunyan`
TEST_LOG=true cargo test health_check_works | bunyan
```

and sift through the output to understand what is going on.
Neat, isn't it?

4.5.12 Cleaning Up Instrumentation Code - `tracing::instrument`

We refactored our initialisation logic. Let's have a look at our instrumentation code now.
Time to bring `subscribe` back once again.

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let request_id = Uuid::new_v4();
    let request_span = tracing::info_span!(
        "Adding a new subscriber",
        %request_id,
        subscriber_email = %form.email,
        subscriber_name= %form.name
    );
    let _request_span_guard = request_span.enter();
    let query_span = tracing::info_span!(
        "Saving new subscriber details in the database"
    );
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .instrument(query_span)
        .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(e) => {
            tracing::error!("Failed to execute query: {:?}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

It is fair to say logging has added some noise to our `subscribe` function.
Let's see if we can cut it down a bit.

We will start with `request_span`: we'd like all operations within `subscribe` to happen within the context of `request_span`.
In other words, we'd like to *wrap* the `subscribe` function in a span.
This requirement is fairly common: extracting each sub-task in its own function is a common way to structure routines to improve readability and make it easier to write tests; therefore we will often want to *attach* a span to a function declaration.

`tracing` caters for this specific usecase with its `tracing::instrument` procedural macro. Let's see it in action:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument(
    name = "Adding a new subscriber",
    skip(form, pool),
    fields(
        request_id = %Uuid::new_v4(),
        subscriber_email = %form.email,
        subscriber_name= %form.name
    )
)]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let query_span = tracing::info_span!(
        "Saving new subscriber details in the database"
    );
    match sqlx::query!(/* */)
        .execute(pool.get_ref())
        .instrument(query_span)
        .await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(e) => {
            tracing::error!("Failed to execute query: {:?}", e);
            HttpResponse::InternalServerError().finish()
        }
    }
}
```

`#[tracing::instrument]` creates a span at the beginning of the function invocation and automatically attaches all arguments passed to the function to the context of the span - in our case, `form` and `pool`. Often function arguments won't be displayable on log records (e.g. `pool`) or we'd like to specify more explicitly what should/how they should be captured (e.g. naming each field of `form`) - we can explicitly tell `tracing` to ignore them using the `skip` directive.

`name` can be used to specify the message associated to the function span - if omitted, it defaults to the function name.

We can also enrich the span's context using the `fields` directive. It leverages the same syntax we have already seen for the `info_span!` macro.

The result is quite nice: all instrumentation concerns are visually separated by execution concerns - the first are dealt with in a procedural macro that "decorates" the function declaration, while the function body focuses on the actual business logic.

It is important to point out that `tracing::instrument` takes care as well to use `Instrument::instrument` if it is applied to an asynchronous function.

Let's extract the query in its own function and use `tracing::instrument` to get rid of `query_span` and the call to the `.instrument` method:

```
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument(
    name = "Adding a new subscriber",
    skip(form, pool),
    fields(
        request_id = %Uuid::new_v4(),
        subscriber_email = %form.email,
```

```rust
        subscriber_name= %form.name
    )
)]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    match insert_subscriber(&pool, &form).await
    {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish()
    }
}

#[tracing::instrument(
    name = "Saving new subscriber details in the database",
    skip(form, pool)
)]
pub async fn insert_subscriber(
    pool: &PgPool,
    form: &FormData,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
    INSERT INTO subscriptions (id, email, name, subscribed_at)
    VALUES ($1, $2, $3, $4)
            "#,
        Uuid::new_v4(),
        form.email,
        form.name,
        Utc::now()
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    // Using the `?` operator to return early
    // if the function failed, returning a sqlx::Error
    // We will talk about error handling in depth later!
    })?;
    Ok(())
}
```

The error event does now fall within the query span and we have a better separation of concerns:

- `insert_subscriber` takes care of the database logic and it has no awareness of the surrounding web framework - i.e. we are not passing `web::Form` or `web::Data` wrappers as input types;
- `subscribe` orchestrates the work to be done by calling the required routines and translates their outcome into the proper response according to the rules and conventions of the HTTP protocol.

I must confess my unbounded love for `tracing::instrument`: it significantly lowers the effort required to instrument your code.

It pushes you in the **pit of success**: the right thing to do *is* the easiest thing to do.

4.5.13 Protect Your Secrets - `secrecy`

There is actually one element of `#[tracing::instrument]` that I am not fond of: it automatically attaches all arguments passed to the function to the context of the span - you have to **opt-out** of logging function inputs (via `skip`) rather than **opt-in**[6].

You do not want secrets (e.g. a password) or personal identifiable information (e.g. the billing address of an end user) in your logs.
Opt-out is a dangerous default - every time you add a new input to a function using `#[tracing::instrument]` you need to ask yourself: is it *safe* to log this? Should I `skip` it?
Give it enough time and somebody will forget - you now have a security incident to deal with[7].

You can prevent this scenario by introducing a wrapper type that **explicitly** marks which fields are considered to be sensitive - `secrecy::Secret`.

```
#! Cargo.toml
# [...]
[dependencies]
secrecy = { version = "0.8", features = ["serde"] }
# [...]
```

Let's check out its definition:

```
/// Wrapper type for values that contains secrets, which attempts to limit
/// accidental exposure and ensure secrets are wiped from memory when dropped.
/// (e.g. passwords, cryptographic keys, access tokens or other credentials)
///
/// Access to the secret inner value occurs through the [...]
/// `expose_secret()` method [...]
pub struct Secret<S>
    where
        S: Zeroize,
{
    /// Inner secret value
    inner_secret: S,
}
```

Memory wiping, provided by the `Zeroize` trait, is a nice-to-have.
The key property we are looking for is `Secret`'s masked `Debug` implementation: `println!("{:?}", my_secret_string)` outputs `Secret([REDACTED String])` instead of the actual secret value. This is exactly what we need to prevent accidental leakage of sensitive material via `#[tracing::instrument]` or other logging statements.
There is an additional upside to an explicit wrapper type: it serves as documentation for new developers who are being introduced to the codebase. It nails down what is considered sensitive in your domain/according to the relevant regulation.

[6] There is a chance that `tracing`'s default behaviour will be changed to be opt-in rather than opt-out in the next breaking release (`0.2.x`).

[7] Some of these security incidents are pretty severe - e.g. Facebook logged by mistake hundreds of millions of plaintext passwords.

The only secret value we need to worry about, right now, is the database password. Let's wrap it up:

```rust
//! src/configuration.rs
use secrecy::Secret;
// [..]

#[derive(serde::Deserialize)]
pub struct DatabaseSettings {
    // [...]
    pub password: Secret<String>,
}
```

`Secret` does not interfere with deserialization - `Secret` implements `serde::Deserialize` by delegating to the deserialization logic of the wrapped type (if you enable the `serde` feature flag, as we did).
The compiler is not happy:

```
error[E0277]: `Secret<std::string::String>` doesn't implement `std::fmt::Display`
--> src/configuration.rs:29:28
|
|             self.username, self.password, self.host, self.port
|                            ^^^^^^^^^^^^^
| `Secret<std::string::String>` cannot be formatted with the default formatter
```

That is a feature, not a bug - `secret::Secret` does not implement `Display` therefore we need to explicitly allow the exposure of the wrapped secret. The compiler error is a great prompt to notice that the entire database connection string should be marked as `Secret` as well given that it embeds the database password:

```rust
//! src/configuration.rs
use secrecy::ExposeSecret;
// [...]

impl DatabaseSettings {
    pub fn connection_string(&self) -> Secret<String> {
        Secret::new(format!(
            "postgres://{}:{}@{}:{}/{}",
            // [...]
            self.password.expose_secret(),
            // [...]
        ))
    }

    pub fn connection_string_without_db(&self) -> Secret<String> {
        Secret::new(format!(
            "postgres://{}:{}@{}:{}",
            // [...]
            self.password.expose_secret(),
            // [...]
        ))
    }
}
```

```rust
//! src/main.rs
use secrecy::ExposeSecret;
// [...]
```

```rust
#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let connection_pool =
        PgPool::connect(&configuration.database.connection_string().expose_secret())
            .await
            .expect("Failed to connect to Postgres.");
    // [...]
}
```

```rust
//! tests/health_check.rs
use secrecy::ExposeSecret;
// [...]

pub async fn configure_database(config: &DatabaseSettings) -> PgPool {
    let mut connection =
        PgConnection::connect(&config.connection_string_without_db().expose_secret())
            .await
            .expect("Failed to connect to Postgres");
    // [...]
    let connection_pool = PgPool::connect(&config.connection_string().expose_secret())
        .await
        .expect("Failed to connect to Postgres.");
    // [...]
}
```

This is it for the time being - going forward we will make sure to wrap sensitive values into `Secret` as soon as they are introduced.

4.5.14 Request Id

We have one last job to do: ensure all logs for a particular request, in particular the record with the returned status code, are enriched with a `request_id` property. How?

If our goal is to avoid touching `actix_web::Logger` the easiest solution is adding another middleware, `RequestIdMiddleware`, that is in charge of:

- generating a unique request identifier;
- creating a new span with the request identifier attached as context;
- wrapping the rest of the middleware chain in the newly created span.

We would be leaving a lot on the table though: `actix_web::Logger` does not give us access to its rich information (status code, processing time, caller IP, etc.) in the same structured JSON format we are getting from other logs - we would have to parse all that information out of its message string.
We are better off, in this case, by bringing in a solution that is `tracing`-aware.

Let's add `tracing-actix-web` as one of our dependencies[8]:

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

[8] Full disclosure - I am the author of `tracing-actix-web`.

```
tracing-actix-web = "0.5"
# [...]
```

It is designed as a drop-in replacement of `actix-web`'s `Logger`, just based on `tracing` instead of `log`:

```rust
//! src/startup.rs
use crate::routes::{health_check, subscribe};
use actix_web::dev::Server;
use actix_web::web::Data;
use actix_web::{web, App, HttpServer};
use sqlx::PgPool;
use std::net::TcpListener;
use tracing_actix_web::TracingLogger;

pub fn run(listener: TcpListener, db_pool: PgPool) -> Result<Server, std::io::Error> {
    let db_pool = Data::new(db_pool);
    let server = HttpServer::new(move || {
        App::new()
            // Instead of `Logger::default`
            .wrap(TracingLogger::default())
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            .app_data(db_pool.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

If you launch the application and fire a request you should see a `request_id` on all logs as well as `request_path` and a few other useful bits of information.

We are almost done - there is one outstanding issue we need to take care of.
Let's take a closer look at the emitted log records for a `POST /subscriptions` request:

```
{
    "msg": "[REQUEST - START]",
    "request_id": "21fec996-ace2-4000-b301-263e319a04c5",
    ...
}
{
    "msg": "[ADDING A NEW SUBSCRIBER - START]",
    "request_id":"aaccef45-5a13-4693-9a69-5",
    ...
}
```

We have two different `request_id` for the same request!
The bug can be traced back to the `#[tracing::instrument]` annotation on our `subscribe` function:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument(
    name = "Adding a new subscriber",
    skip(form, pool),
    fields(
        request_id = %Uuid::new_v4(),
        subscriber_email = %form.email,
        subscriber_name= %form.name
    )
)]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    // [...]
}

// [...]
```

We are still generating a `request_id` at the function-level which overrides the `request_id` coming from `TracingLogger`.
Let's get rid of it to fix the issue:

```
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument(
    name = "Adding a new subscriber",
    skip(form, pool),
    fields(
        subscriber_email = %form.email,
        subscriber_name= %form.name
    )
)]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    // [...]
}

// [...]
```

All good now - we have one consistent `request_id` for each endpoint of our application.

4.5.15 Leveraging The `tracing` Ecosystem

We covered a lot of what `tracing` has to offer - it has significantly improved the quality of the telemetry data we are collecting as well as the clarity of our instrumentation code.
At the same time, we have barely touched upon the richness of the whole `tracing` ecosystem when it comes to subscriber layers.
Just to mention a few more of those readily available:

- `tracing-actix-web` is OpenTelemetry-compatible. If you plug-in `tracing-opentelemetry` you can ship spans to an OpenTelemetry-compatible service (e.g. Jaeger or Honeycomb.io) for further analysis;
- `tracing-error` enriches our error types with a `SpanTrace` to ease troubleshooting.

It is not an exaggeration to state that `tracing` is a foundational crate in the Rust ecosystem. While `log` is the minimum common denominator, `tracing` is now established as the modern backbone of the whole diagnostics and instrumentation ecosystem.

4.6 Summary

We started from a completely silent `actix-web` application and we ended up with high-quality telemetry data. It is now time to take this newsletter API live!

In the next chapter we will build a basic deployment pipeline for our Rust project.

Chapter 5

Going Live

We have a working prototype of our newsletter API - it is now time to take it **live**.
We will learn how to package our Rust application as a Docker container to deploy it on DigitalOcean's App Platform.
At the end of the chapter we will have a **C**ontinuous **D**eployment (CD) pipeline: every commit to the `main` branch will *automatically* trigger the deployment of the latest version of the application to our users.

5.1 We Must Talk About Deployments

Everybody loves to talk about how important it is to deploy software to production as often as possible (and I put myself in that bunch!).

> "Get customer feedback early!"
> "Ship often and iterate on the product!"

But nobody shows you *how*.

Pick a random book on web development or an introduction to framework XYZ.
Most will not dedicate more than a paragraph to the topic of deployments.
A few will have a chapter about it - usually towards the end of the book, the part you never get to actually read.
A handful actually give it the space it deserves, as early as they reasonably can.

Why?

Because deployments are (still) a messy business.
There are many vendors, most are not straight-forward to use and what is considered state-of-art or best-practice tends to change really quickly[1].
That is why most authors steer away from the topic: it takes many pages and it is painful to write something down to realise, one or two years later, that it is already out of date.

Nonetheless deployments are a prominent concern in the daily life of a software engineer - e.g. it is difficult to talk about database schema migrations, domain validation and API evolution without

[1] Kubernetes is six years old, Docker itself is just seven years old!

taking into account your deployment process.
We simply cannot ignore the topic in a book called *Zero To Production*.

5.2 Choosing Our Tools

The purpose of this chapter is to get you to experience, first hand, what it means to *actually* deploy on every commit to your `main` branch.
That is why we are talking about deployment as early as chapter five: to give you the chance to practice this muscle for the rest of the book, as you would actually be doing if this was a real commercial project.
We are particularly interested, in fact, on how the engineering practice of continuous deployment influences our design choices and development habits.

At the same time, building the perfect continuous deployment pipeline is not the focus of the book - it deserves a book on its own, probably a whole company.
We have to be pragmatic and strike a balance between intrinsic usefulness (i.e. learn a tool that is valued in the industry) and developer experience.
And even if we spent the time to hack together the "best" setup, you are still likely to end up choosing different tools and different vendors due to the specific constraints of your organisation.

What matters is the underlying *philosophy* and getting you to try continuous deployment as a practice.

5.2.1 Virtualisation: Docker

Our local development environment and our production environment serve two very different purposes.
Browsers, IDEs, our music playlists - they can co-exist on our local machine. It is a multi-purpose workstation.
Production environments, instead, have a much narrower focus: running our software to make it available to our users. Anything that is not strictly related to that goal is either a waste of resources, at best, or a security liability, at worst.

This discrepancy has historically made deployments fairly troublesome, leading to the now memefied complaint "It works on my machine!".
It is not enough to copy the source code to our production servers. Our software is likely to make assumptions on the capabilities exposed by the underlying operating system (e.g. a native Windows application will not run on Linux), on the availability of other software on the same machine (e.g. a certain version of the Python interpreter) or on its configuration (e.g. do I have root permissions?).
Even if we started with two identical environments we would, over time, run into troubles as versions drift and subtle inconsistencies come up to haunt our nights and weekends.

The easiest way to ensure that our software runs correctly is to tightly control the *environment* it is being executed into.

This is the fundamental idea behind virtualisation technology: what if, instead of shipping code to production, you could ship a self-contained environment that included your application?!
It would work great for both sides: less Friday-night surprises for you, the developer; a consistent abstraction to build on top of for those in charge of the production infrastructure.
Bonus points if the environment itself can be specified as code to ensure reproducibility.

The nice thing about virtualisation is that it exists and it has been mainstream for almost a decade now.
As for most things in technology, you have a few options to choose from depending on your needs: virtual machines, containers (e.g. Docker) and a few others (e.g. Firecracker).

We will go with the mainstream and ubiquitous option - Docker containers.

5.2.2 Hosting: DigitalOcean

AWS, Google Cloud, Azure, Digital Ocean, Clever Cloud, Heroku, Qovery…
The list of vendors you can pick from to host your software goes on and on.
People have made a successful business out of recommending the best cloud tailored to your specific needs and usecases - not my job (yet) or the purpose of this book.

We are looking for something that is easy to use (great developer experience, minimal unnecessary complexity) and fairly established.
In November 2020, the intersection of those two requirements seems to be Digital Ocean, in particular their newly launched App Platform proposition.

> Disclaimer: Digital Ocean is not paying me to promote their services here.

5.3 A Dockerfile For Our Application

DigitalOcean's App Platform has native support for deploying containerised applications.
This is going to be our first task: we have to write a Dockerfile to build and execute our application as a Docker container.

5.3.1 Dockerfiles

A Dockerfile is a *recipe* for your application environment.
They are organised in layers: you start from a base *image* (usually an OS enriched with a programming language toolchain) and execute a series of commands (`COPY`, `RUN`, etc.), one after the other, to build the environment you need.

Let's have a look at the simplest possible Dockerfile for a Rust project:

```
# We use the latest Rust stable release as base image
FROM rust:1.59.0

# Let's switch our working directory to `app` (equivalent to `cd app`)
# The `app` folder will be created for us by Docker in case it does not
# exist already.
WORKDIR /app
# Install the required system dependencies for our linking configuration
RUN apt update && apt install lld clang -y
# Copy all files from our working environment to our Docker image
COPY . .
# Let's build our binary!
# We'll use the release profile to make it faaaast
RUN cargo build --release
```

```
# When `docker run` is executed, launch the binary!
ENTRYPOINT ["./target/release/zero2prod"]
```

Save it in a file named `Dockerfile` in the root directory of our git repository:

```
zero2prod/
  .github/
  migrations/
  scripts/
  src/
  tests/
  .gitignore
  Cargo.lock
  Cargo.toml
  configuration.yaml
  Dockerfile
```

The process of executing those commands to get an image is called *building*.
Using the Docker CLI:

```
# Build a docker image tagged as "zero2prod" according to the recipe
# specified in `Dockerfile`
docker build --tag zero2prod --file Dockerfile .
```

What does the `.` at the end of the command stand for?

5.3.2 Build Context

`docker build` generates an image starting from a recipe (the Dockerfile) and a *build context*.
You can picture the Docker image you are building as its own fully isolated environment.
The only point of contact between the image and your local machine are commands like `COPY` or `ADD`[2]: the build context determines what files on your host machine are visible inside the Docker container to `COPY` and its friends.

Using `.` we are telling Docker to use the current directory as the build context for this image; `COPY . app` will therefore copy all files from the current directory (including our source code!) into the `app` directory of our Docker image.
Using `.` as build context implies, for example, that Docker will not allow `COPY` to see files from the parent directory or from arbitrary paths on your machine into the image.

You could use a different path or even a URL (!) as build context depending on your needs.

5.3.3 Sqlx Offline Mode

If you were eager enough, you might have already launched the build command… just to realise it doesn't work!

```
docker build --tag zero2prod --file Dockerfile .
```

```
# [...]
Step 4/5 : RUN cargo build --release
# [...]
```

[2] Unless you are using `--network=host`, `--ssh` or other similar options. You also have volumes as an alternative mechanism to share files at runtime.

```
error: error communicating with the server:
Cannot assign requested address (os error 99)
  --> src/routes/subscriptions.rs:35:5
   |
35 | /     sqlx::query!(
36 | |         r#"
37 | |     INSERT INTO subscriptions (id, email, name, subscribed_at)
38 | |     VALUES ($1, $2, $3, $4)
...  |
43 | |         Utc::now()
44 | |     )
   | |_____^
   |
   = note: this error originates in a macro
```

What is going on?
`sqlx` calls into our database at compile-time to ensure that all queries can be successfully executed considering the schemas of our tables.
When running `cargo build` inside our Docker image, though, `sqlx` fails to establish a connection with the database that the `DATABASE_URL` environment variable in the `.env` file points to.

How do we fix it?

We could allow our image to talk to a database running on our local machine at build time using the `--network` flag. This is the strategy we follow in our CI pipeline given that we need the database anyway to run our integration tests.
Unfortunately it is somewhat troublesome to pull off for Docker builds due to how Docker networking is implemented on different operating systems (e.g. MacOS) and would significantly compromise how reproducible our builds are.

A better option is to use the newly-introduced offline mode for `sqlx`.
Let's add the `offline` feature to `sqlx` in our `Cargo.toml`:

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

# Using table-like toml syntax to avoid a super-long line!
[dependencies.sqlx]
version = "0.5.7"
default-features = false
features = [
    "runtime-actix-rustls",
    "macros",
    "postgres",
    "uuid",
    "chrono",
    "migrate",
    "offline"
]
```

The next step relies on `sqlx`'s CLI. The command we are looking for is `sqlx prepare`. Let's look at its help message:

```
sqlx prepare --help
```

```
sqlx-prepare
Generate query metadata to support offline compile-time verification.

Saves metadata for all invocations of `query!` and related macros to
`sqlx-data.json` in the current directory, overwriting if needed.

During project compilation, the absence of the `DATABASE_URL` environment
variable or the presence of `SQLX_OFFLINE` will constrain the compile-time
verification to only read from the cached query metadata.

USAGE:
    sqlx prepare [FLAGS] [-- <args>...]

ARGS:
    <args>...
            Arguments to be passed to `cargo rustc ...`

FLAGS:
        --check
            Run in 'check' mode. Exits with 0 if the query metadata is up-to-date.
            Exits with 1 if the query metadata needs updating
```

In other words, `prepare` performs the same work that is usually done when `cargo build` is invoked but it saves the outcome of those queries to a metadata file (`sqlx-data.json`) which can later be detected by `sqlx` itself and used to skip the queries altogether and perform an offline build.

Let's invoke it!

```
# It must be invoked as a cargo subcommand
# All options after `--` are passed to cargo itself
# We need to point it at our library since it contains
# all our SQL queries.
cargo sqlx prepare -- --lib
```

```
query data written to `sqlx-data.json` in the current directory;
please check this into version control
```

We will indeed commit the file to version control, as the command output suggests.
Let's set the `SQLX_OFFLINE` environment variable to `true` in our Dockerfile to force `sqlx` to look at the saved metadata instead of trying to query a live database:

```
FROM rust:1.59.0

WORKDIR /app
RUN apt update && apt install lld clang -y
COPY . .
ENV SQLX_OFFLINE true
RUN cargo build --release
ENTRYPOINT ["./target/release/zero2prod"]
```

Let's try again to build our Docker container:

```
docker build --tag zero2prod --file Dockerfile .
```

There should be no errors this time!

We have a problem though: how do we ensure that `sqlx-data.json` does not go out of sync (e.g. when the schema of our database changes or when we add new queries)?

We can use the `--check` flag in our CI pipeline to ensure that it stays up-to-date - check the updated pipeline definition in the book GitHub repository as a reference.

5.3.4 Running An Image

When building our image we attached a tag to it, `zero2prod`:

```
docker build --tag zero2prod --file Dockerfile .
```

We can use the tag to refer to the image in other commands. In particular, to *run it*:

```
docker run zero2prod
```

`docker run` will trigger the execution of the command we specified in our `ENTRYPOINT` statement:

```
ENTRYPOINT ["./target/release/zero2prod"]
```

In our case, it will execute our binary therefore launching our API.
Let's launch our image then!

You should immediately see an error:

```
thread 'main' panicked at
  'Failed to connect to Postgres:
   Io(Os {
     code: 99,
     kind: AddrNotAvailable,
     message: "Cannot assign requested address"
   })'
```

This is coming from this line in our `main` function:

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let connection_pool = PgPool::connect(
            &configuration.database.connection_string().expose_secret()
        )
        .await
        .expect("Failed to connect to Postgres.");
    // [...]
}
```

We can relax our requirements by using `connect_lazy` - it will only try to establish a connection when the pool is used for the first time.

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    // No longer async, given that we don't actually try to connect!
    let connection_pool = PgPool::connect_lazy(
            &configuration.database.connection_string().expose_secret()
        )
        .expect("Failed to create Postgres connection pool.");
    // [...]
}
```

We can now re-build the Docker image and run it again: you should immediately see a couple of log lines! Let's open another terminal and try to make a request to our health check endpoint:

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

```
curl: (7) Failed to connect to 127.0.0.1 port 8000: Connection refused
```

Not great.

5.3.5 Networking

By default, Docker images do not expose their ports to the underlying host machine. We need to do it explicitly using the `-p` flag.
Let's kill our running image to launch it again using:

```
docker run -p 8000:8000 zero2prod
```

Trying to hit the health check endpoint will trigger the same error message.
We need to dig into our `main.rs` file to understand why:

```
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::run;
use zero2prod::telemetry::{get_subscriber, init_subscriber};
use sqlx::postgres::PgPool;
use std::net::TcpListener;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into(), std::io::stdout);
    init_subscriber(subscriber);

    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPool::connect_lazy(
            &configuration.database.connection_string().expose_secret()
        )
        .expect("Failed to create Postgres connection pool.");

    let address = format!("127.0.0.1:{}", configuration.application_port);
    let listener = TcpListener::bind(address)?;
    run(listener, connection_pool)?.await?;
    Ok(())
}
```

We are using `127.0.0.1` as our host in `address` - we are instructing our application to only accept connections coming from the same machine.
However, we are firing a GET request to `/health_check` from the host machine, which is not seen as local by our Docker image, therefore triggering the `Connection refused` error we have

just seen.

We need to use `0.0.0.0` as host to instruct our application to accept connections from any network interface, not just the local one.
We should be careful though: using `0.0.0.0` significantly increases the "audience" of our application, with some security implications.
The best way forward is to make the host portion of our `address` configurable - we will keep using `127.0.0.1` for our local development and set it to `0.0.0.0` in our Docker images.

5.3.6 Hierarchical Configuration

Our `Settings` struct currently looks like this:

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

#[derive(serde::Deserialize)]
pub struct Settings {
    pub database: DatabaseSettings,
    pub application_port: u16,
}

#[derive(serde::Deserialize)]
pub struct DatabaseSettings {
    pub username: String,
    pub password: Secret<String>,
    pub port: u16,
    pub host: String,
    pub database_name: String,
}

// [...]
```

Let's introduce another struct, `ApplicationSettings`, to group together all configuration values related to our application address:

```rust
#[derive(serde::Deserialize)]
pub struct Settings {
    pub database: DatabaseSettings,
    pub application: ApplicationSettings,
}

#[derive(serde::Deserialize)]
pub struct ApplicationSettings {
    pub port: u16,
    pub host: String,
}

// [...]
```

We need to update our `configuration.yml` file to match the new structure:

```yaml
#! configuration.yml
application:
  port: 8000
  host: 127.0.0.1
```

```yaml
database:
  # [...]
```

as well as our `main.rs`, where we will leverage the new configurable `host` field:

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let address = format!(
        "{}:{}",
        configuration.application.host, configuration.application.port
    );
    // [...]
}
```

The host is now read from configuration, but how do we use a different value for different environments?
We need to make our configuration *hierarchical*.

Let's have a look at `get_configuration`, the function in charge of loading our `Settings` struct:

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

pub fn get_configuration() -> Result<Settings, config::ConfigError> {
    let mut settings = config::Config::default();

    settings.merge(config::File::with_name("configuration"))?;

    settings.try_into()
}
```

We are reading from a file named `configuration` to populate `Settings`'s fields. There is no further room for tuning the values specified in our `configuration.yaml`.
Let's take a more refined approach. We will have:

- A base configuration file, for values that are shared across our local and production environment (e.g. database name);
- A collection of environment-specific configuration files, specifying values for fields that require customisation on a per-environment basis (e.g. host);
- An environment variable, `APP_ENVIRONMENT`, to determine the running environment (e.g. `production` or `local`).

All configuration files will live in the same top-level directory, `configuration`.

The good news is that `config`, the crate we are using, supports all the above out of the box!
Let's put it together:

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

pub fn get_configuration() -> Result<Settings, config::ConfigError> {
    let mut settings = config::Config::default();
    let base_path = std::env::current_dir().expect("Failed to determine the current directory");
    let configuration_directory = base_path.join("configuration");

    // Read the "default" configuration file
    settings.merge(config::File::from(configuration_directory.join("base")).required(true))?;

    // Detect the running environment.
    // Default to `local` if unspecified.
    let environment: Environment = std::env::var("APP_ENVIRONMENT")
        .unwrap_or_else(|_| "local".into())
        .try_into()
        .expect("Failed to parse APP_ENVIRONMENT.");

    // Layer on the environment-specific values.
    settings.merge(
        config::File::from(configuration_directory.join(environment.as_str())).required(true),
    )?;

    settings.try_into()
}

/// The possible runtime environment for our application.
pub enum Environment {
    Local,
    Production,
}

impl Environment {
    pub fn as_str(&self) -> &'static str {
        match self {
            Environment::Local => "local",
            Environment::Production => "production",
        }
    }
}

impl TryFrom<String> for Environment {
    type Error = String;

    fn try_from(s: String) -> Result<Self, Self::Error> {
        match s.to_lowercase().as_str() {
            "local" => Ok(Self::Local),
            "production" => Ok(Self::Production),
            other => Err(format!(
                "{} is not a supported environment. Use either `local` or `production`.",
                other
            )),
        }
    }
}
```

Let's refactor our configuration file to match the new structure.
We have to get rid of `configuration.yaml` and create a new `configuration` directory with `base.yaml`, `local.yaml` and `production.yaml` inside.

```yaml
#! configuration/base.yaml
application:
  port: 8000
database:
  host: "localhost"
  port: 5432
  username: "postgres"
  password: "password"
  database_name: "newsletter"
```

```yaml
#! configuration/local.yaml
application:
  host: 127.0.0.1
```

```yaml
#! configuration/production.yaml
application:
  host: 0.0.0.0
```

We can now instruct the binary in our Docker image to use the production configuration by setting the `APP_ENVIRONMENT` environment variable with an `ENV` instruction:

```dockerfile
FROM rust:1.59.0
WORKDIR /app
RUN apt update && apt install lld clang -y
COPY . .
ENV SQLX_OFFLINE true
RUN cargo build --release
ENV APP_ENVIRONMENT production
ENTRYPOINT ["./target/release/zero2prod"]
```

Let's rebuild our image and launch it again:

```bash
docker build --tag zero2prod --file Dockerfile .
docker run -p 8000:8000 zero2prod
```

One of the first log lines should be something like

```json
{
    "name":"zero2prod",
    "msg":"Starting \"actix-web-service-0.0.0.0:8000\" service on 0.0.0.0:8000",
    ...
}
```

If it is, good news - our configuration works as expected!

Let's try again to hit the health check endpoint:

```bash
curl -v http://127.0.0.1:8000/health_check
```

```
curl -v http://127.0.0.1:8000/health_check

> GET /health_check HTTP/1.1
> Host: 127.0.0.1:8000
> User-Agent: curl/7.61.0
> Accept: */*
>
< HTTP/1.1 200 OK
< content-length: 0
< date: Sun, 01 Nov 2020 17:32:19 GMT
```

It works, awesome!

5.3.7 Database Connectivity

What about `POST /subscriptions`?

```
curl --request POST \
  --data 'name=le%20guin&email=ursula_le_guin%40gmail.com' \
  127.0.0.1:8000/subscriptions --verbose
```

A long wait, then a 500!
Let's look at the application logs (useful, aren't they?)

```
{
    "msg": "[SAVING NEW SUBSCRIBER DETAILS IN THE DATABASE - EVENT] \
            Failed to execute query: PoolTimedOut",
    ...
}
```

This should not come as a surprise - we swapped `connect` with `connect_lazy` to avoid dealing with the database straight away.

It took us half a minute to see a 500 coming back - that is because 30 seconds is the default timeout to acquire a connection from the pool in `sqlx`.
Let's fail a little faster by using a shorter timeout:

```rust
//! src/main.rs
use sqlx::postgres::PgPoolOptions;
// [...]

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let connection_pool = PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        .connect_lazy(
            &configuration.database.connection_string().expose_secret()
        )
        .expect("Failed to create Postgres connection pool.");
    // [...]
}
```

There are various ways to get a working local setup using Docker containers:

- Run the application container with `--network=host`, as we are currently doing for the Postgres container;
- Use `docker-compose`;
- Create a user-defined network.

A working local setup does not get us any closer to having a working database connection when deployed on Digital Ocean. We will therefore let it be for now.

5.3.8 Optimising Our Docker Image

As far as our Docker image is concerned, it seems to work as expected - time to deploy it!
Well, not yet.

There are two optimisations we can make to our Dockerfile to make our life easier going forward:

- smaller image size for faster usage;
- Docker layer caching for faster builds.

5.3.8.1 Docker Image Size

We will not be running `docker build` on the machines hosting our application. They will be using `docker pull` to *download* our Docker image without going through the process of building it from scratch.

This is extremely convenient: it can take quite a long time to build our image (and it certainly does in Rust!) and we only need to pay that cost once.
To actually use the image we only need to pay for its download cost which is directly related to its *size*.

How big is our image?

We can find out using

```
docker images zero2prod
```

```
REPOSITORY          TAG             SIZE
zero2prod           latest          2.31GB
```

Is that big or small?
Well, our final image cannot be any smaller than the image we used as base - `rust:1.59.0`. How big is that?

```
docker images rust:1.59.0
```

```
REPOSITORY          TAG             SIZE
rust                1.59.0            1.29GB
```

Ok, our final image is almost twice as heavy as our base image.
We can do much better than that!

Our first line of attack is reducing the size of the Docker build context by excluding files that are not needed to build our image.
Docker looks for a specific file in our project to determine what should be ignored - `.dockerignore`
Let's create one in the root directory with the following content:

```
.env
target/
tests/
Dockerfile
scripts/
migrations/
```

All files that match the patterns specified in `.dockerignore` are not sent by Docker as part of the build context to the image, which means they will not be in scope for `COPY` instructions.

This will massively speed up our builds (and reduce the size of the final image) if we get to ignore heavy directories (e.g. the `target` folder for Rust projects).

The next optimisation, instead, leverages one of Rust's unique strengths.
Rust's binaries are statically linked[3] - we do not need to keep the source code or intermediate compilation artifacts around to run the binary, it is entirely self-contained.
This plays nicely with *multi-stage* builds, a useful Docker feature. We can split our build in two stages:

- a `builder` stage, to generate a compiled binary;
- a `runtime` stage, to run the binary.

The modified Dockerfile looks like this:

```
# Builder stage
FROM rust:1.59.0 AS builder

WORKDIR /app
RUN apt update && apt install lld clang -y
COPY . .
ENV SQLX_OFFLINE true
RUN cargo build --release

# Runtime stage
FROM rust:1.59.0 AS runtime

WORKDIR /app
# Copy the compiled binary from the builder environment
# to our runtime environment
COPY --from=builder /app/target/release/zero2prod zero2prod
# We need the configuration file at runtime!
COPY configuration configuration
ENV APP_ENVIRONMENT production
ENTRYPOINT ["./zero2prod"]
```

`runtime` is our final image.
The `builder` stage does not contribute to its size - it is an intermediate step and it is discarded at the end of the build. The only piece of the `builder` stage that is found in the final artifact is what we explicitly copy over - the compiled binary!

What is the image size using the above Dockerfile?

```
docker images zero2prod
```

```
REPOSITORY          TAG              SIZE
zero2prod           latest           1.3GB
```

Just 20 MBs bigger than the size of our base image, much better!
We can go one step further: instead of using `rust:1.59.0` for our `runtime` stage we can switch to `rust:1.59.0-slim`, a smaller image using the same underlying OS.

```
# [...]
# Runtime stage
```

[3] `rustc` statically links all Rust code but dynamically links `libc` from the underlying system if you are using the Rust standard library. You can get a fully statically linked binary by targeting `linux-musl`, see here.

```
FROM rust:1.59.0-slim AS runtime
# [ ... ]
```

```
docker images zero2prod
```

```
REPOSITORY          TAG             SIZE
zero2prod           latest          681MB
```

That is 4x smaller than what we had at the beginning - not bad at all!
We can go even smaller by shaving off the weight of the whole Rust toolchain and machinery (i.e. `rustc`, `cargo`, etc) - none of that is needed to *run* our binary.
We can use the bare operating system as base image (`debian:bullseye-slim`) for our runtime stage:

```
# [ ... ]
# Runtime stage
FROM debian:bullseye-slim AS runtime
WORKDIR /app
# Install OpenSSL - it is dynamically linked by some of our dependencies
# Install ca-certificates - it is needed to verify TLS certificates
# when establishing HTTPS connections
RUN apt-get update -y \
    && apt-get install -y --no-install-recommends openssl ca-certificates \
    # Clean up
    && apt-get autoremove -y \
    && apt-get clean -y \
    && rm -rf /var/lib/apt/lists/*
COPY --from=builder /app/target/release/zero2prod zero2prod
COPY configuration configuration
ENV APP_ENVIRONMENT production
ENTRYPOINT ["./zero2prod"]
```

```
docker images zero2prod
```

```
REPOSITORY          TAG             SIZE
zero2prod           latest          88.1MB
```

Less than a 100 MBs - ~25x smaller than our initial attempt[4].
We could go *even smaller* by using `rust:1.59.0-alpine`, but we would have to cross-compile to the `linux-musl` target - out of scope for now. Check out `rust-musl-builder` if you are interested in generating tiny Docker images.
Another option to reduce the size of our binary further is *stripping* symbols from it - you can find more information about it here.

5.3.8.2 Caching For Rust Docker Builds

Rust shines at runtime, consistently delivering great performance, but it comes at a cost: compilation times. They have been consistently among the top answers in the Rust annual survey when it comes to the biggest challenges or problems for the Rust project.

Optimised builds (`--release`), in particular, can be gruesome - up to 15/20 minutes on medium projects with several dependencies. Quite common on web development projects like ours that are pulling in many foundational crates from the async ecosystem (tokio, actix-web, sqlx, etc.).

[4] Credits to Ian Purton and flat_of_angles for pointing out that there was further room for improvement.

Unfortunately, `--release` is what we use in our `Dockerfile` to get top-performance in our production environment. How can we mitigate the pain?

We can leverage another Docker feature: layer caching.

Each `RUN`, `COPY` and `ADD` instruction in a Dockerfile creates a layer: a diff between the previous state (the layer above) and the current state after having executed the specified command.
Layers are cached: if the starting point of an operation has not changed (e.g. the base image) and the command itself has not changed (e.g. the checksum of the files copied by `COPY`) Docker does not perform any computation and directly retrieves a copy of the result from the local cache.

Docker layer caching is fast and can be leveraged to massively speed up Docker builds.
The trick is optimising the order of operations in your Dockerfile: anything that refers to files that are changing often (e.g. source code) should appear as late as possible, therefore maximising the likelihood of the previous step being unchanged and allowing Docker to retrieve the result straight from the cache.

The expensive step is usually compilation.
Most programming languages follow the same playbook: you `COPY` a lock-file of some kind first, build your dependencies, `COPY` over the rest of your source code and then build your project.
This guarantees that most of the work is cached as long as your dependency tree does not change between one build and the next.

In a Python project, for example, you might have something along these lines:

```
FROM python:3
COPY requirements.txt
RUN pip install -r requirements.txt
COPY src/ /app
WORKDIR /app
ENTRYPOINT ["python", "app"]
```

`cargo`, unfortunately, does not provide a mechanism to build your project dependencies starting from its `Cargo.lock` file (e.g. `cargo build --only-deps`).
Once again, we can rely on a community project to expand `cargo`'s default capability: `cargo-chef`[5].

Let's modify our Dockerfile as suggested in `cargo-chef`'s `README`:

```
FROM lukemathwalker/cargo-chef:latest-rust-1.59.0 as chef
WORKDIR /app
RUN apt update && apt install lld clang -y

FROM chef as planner
COPY . .
# Compute a lock-like file for our project
RUN cargo chef prepare  --recipe-path recipe.json

FROM chef as builder
COPY --from=planner /app/recipe.json recipe.json
# Build our project dependencies, not our application!
RUN cargo chef cook --release --recipe-path recipe.json
# Up to this point, if our dependency tree stays the same,
# all layers should be cached.
```

[5] Full disclosure - I am the author of `cargo-chef`.

```
COPY . .
ENV SQLX_OFFLINE true
# Build our project
RUN cargo build --release --bin zero2prod

FROM debian:bullseye-slim AS runtime
WORKDIR /app
RUN apt-get update -y \
    && apt-get install -y --no-install-recommends openssl ca-certificates \
    # Clean up
    && apt-get autoremove -y \
    && apt-get clean -y \
    && rm -rf /var/lib/apt/lists/*
COPY --from=builder /app/target/release/zero2prod zero2prod
COPY configuration configuration
ENV APP_ENVIRONMENT production
ENTRYPOINT ["./zero2prod"]
```

We are using three stages: the first computes the recipe file, the second caches our dependencies and then builds our binary, the third is our runtime environment. As long as our dependencies do not change the `recipe.json` file will stay the same, therefore the outcome of `cargo chef cook --release --recipe-path recipe.json` will be cached, massively speeding up our builds.

We are taking advantage of how Docker layer caching interacts with multi-stage builds: the `COPY . .` statement in the `planner` stage will invalidate the cache for the `planner` container, but it will not invalidate the cache for the `builder` container as long as the checksum of the `recipe.json` returned by `cargo chef prepare` does not change.
You can think of each stage as its own Docker image with its own caching - they only interact with each other when using the `COPY --from` statement.

This will save us a massive amount of time in the next section.

5.4 Deploy To DigitalOcean Apps Platform

We have built a (damn good) containerised version of our application. Let's deploy it now!

5.4.1 Setup

You have to sign up on Digital Ocean's website.
Once you have an account install `doctl`, Digital Ocean's CLI - you can find instructions here.

> Hosting on Digital Ocean's App Platform is not free - keeping our app and its associated database up and running costs roughly 20.00 USD/month.
> I suggest you to destroy the app at the end of each session - it should keep your spend way below 1.00 USD. I spent 0.20 USD while playing around with it to write this chapter!

5.4.2 App Specification

Digital Ocean's App Platform uses a declarative configuration file to let us specify what our application deployment should look like - they call it *App Spec*.

Looking at the reference documentation, as well as some of their examples, we can piece together a first draft of what our App Spec looks like.

Let's put this manifest, `spec.yaml`, at the root of our project directory.

```yaml
#! spec.yaml
name: zero2prod
# Check https://www.digitalocean.com/docs/app-platform/#regional-availability
# for a list of all the available options.
# You can get region slugs from
# https://www.digitalocean.com/docs/platform/availability-matrix/
# They must specified lowercased.
# `fra` stands for Frankfurt (Germany - EU)
region: fra
services:
  - name: zero2prod
    # Relative to the repository root
    dockerfile_path: Dockerfile
    source_dir: .
    github:
      # Depending on when you created the repository,
      # the default branch on GitHub might have been named `master`
      branch: main
      # Deploy a new version on every commit to `main`!
      # Continuous Deployment, here we come!
      deploy_on_push: true
      # !!! Fill in with your details
      # e.g. LukeMathWalker/zero-to-production
      repo: <YOUR USERNAME>/<YOUR REPOSITORY NAME>
    # Active probe used by DigitalOcean's to ensure our application is healthy
    health_check:
      # The path to our health check endpoint!
      # It turned out to be useful in the end!
      http_path: /health_check
    # The port the application will be listening on for incoming requests
    # It should match what we specified in our configuration/production.yaml file!
    http_port: 8000
    # For production workloads we'd go for at least two!
    # But let's try to keep the bill under control for now...
    instance_count: 1
    instance_size_slug: basic-xxs
    # All incoming requests should be routed to our app
    routes:
      - path: /
```

Take your time to go through all the specified values and understand what they are used for. We can use their CLI, `doctl`, to create the application for the first time:

```
doctl apps create --spec spec.yaml
```

```
Error: Unable to initialize DigitalOcean API client: access token is required.
(hint: run 'doctl auth init')
```

Well, we have to authenticate first.
Let's follow their suggestion:

```
doctl auth init
```

```
Please authenticate doctl for use with your DigitalOcean account.
You can generate a token in the control panel at
https://cloud.digitalocean.com/account/api/tokens
```

Once you have provided your token we can try again:

```
doctl apps create --spec spec.yaml
```

```
Error: POST
https://api.digitalocean.com/v2/apps: 400 GitHub user not
authenticated
```

OK, follow their instructions to link your GitHub account.
Third time's a charm, let's try again!

```
doctl apps create --spec spec.yaml
```

```
Notice: App created
ID         Spec Name    Default Ingress    Active Deployment ID    In Progress Deployment ID
e80 ...    zero2prod
```

It worked!
You can check your app status with

```
doctl apps list
```

or by looking at DigitalOcean's dashboard.

Although the app has been successfully created it is not running yet!
Check the `Deployment` tab on their dashboard - it is probably building the Docker image.
Looking at a few recent issues on their bug tracker it might take a while - more than a few people have reported they experienced slow builds. Digital Ocean's support engineers suggested to leverage Docker layer caching to mitigate the issue - we already covered all the bases there!

> If you experience an out-of-memory error when building your Docker image on DigitalOcean, check out this GitHub issue.

Wait for these lines to show up in their dashboard build logs:

```
zero2prod │ 00:00:20 => Uploaded the built image to the container registry
zero2prod │ 00:00:20 => Build complete
```

Deployed successfully!
You should be able to see the health check logs coming in every ten seconds or so when Digital Ocean's platform pings our application to ensure it is running.

With

```
doctl apps list
```

you can retrieve the public facing URI of your application. Something along the lines of

```
https://zero2prod-aaaaa.ondigitalocean.app
```

Try firing off a health check request now, it should come back with a `200 OK`!

Notice that DigitalOcean took care for us to set up HTTPS by provisioning a certificate and redirecting HTTPS traffic to the port we specified in the application specification. One less thing to worry about.

The `POST /subscriptions` endpoint is still failing, in the very same way it did locally: we do not have a live database backing our application in our production environment.
Let's provision one.

Add this segment to your `spec.yaml` file:

```yaml
databases:
  # PG = Postgres
  - engine: PG
    # Database name
    name: newsletter
    # Again, let's keep the bill lean
    num_nodes: 1
    size: db-s-dev-database
    # Postgres version - using the latest here
    version: "12"
```

Then update your app specification:

```bash
# You can retrieve your app id using `doctl apps list`
doctl apps update YOUR-APP-ID --spec=spec.yaml
```

It will take some time for DigitalOcean to provision a Postgres instance.

In the meantime we need to figure out how to point our application at the database in production.

5.4.3 How To Inject Secrets Using Environment Variables

The connection string will contain values that we do not want to commit to version control - e.g. the username and the password of our database root user.
Our best option is to use environment variables as a way to inject secrets at runtime into the application environment. DigitalOcean's apps, for example, can refer to the `DATABASE_URL` environment variable (or a few others for a more granular view) to get the database connection string at runtime.

We need to upgrade our `get_configuration` function (again) to fulfill our new requirements.

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

pub fn get_configuration() -> Result<Settings, config::ConfigError> {
    let mut settings = config::Config::default();
    let base_path = std::env::current_dir().expect("Failed to determine the current directory");
    let configuration_directory = base_path.join("configuration");
    settings.merge(config::File::from(configuration_directory.join("base")).required(true))?;
    let environment: Environment = std::env::var("APP_ENVIRONMENT")
        .unwrap_or_else(|_| "local".into())
        .try_into()
        .expect("Failed to parse APP_ENVIRONMENT.");
    settings.merge(
        config::File::from(configuration_directory.join(environment.as_str())).required(true),
    )?;

    // Add in settings from environment variables (with a prefix of APP and '__' as separator)
    // E.g. `APP_APPLICATION__PORT=5001 would set `Settings.application.port`
    settings.merge(config::Environment::with_prefix("app").separator("__"))?;

    settings.try_into()
}
```

This allows us to customize **any** value in our `Settings` struct using environment variables, overriding what is specified in our configuration files.

Why is that convenient?
It makes it possible to inject values that are too dynamic (i.e. not known a priori) or too sensitive to be stored in version control.
It also makes it *fast* to change the behaviour of our application: we do not have to go through a full re-build if we want to tune one of those values (e.g. the database port). For languages like Rust, where a fresh build can take ten minutes or more, this can make the difference between a short outage and a substantial service degradation with customer-visible impact.

Before we move on let's take care of an annoying detail: environment variables are strings for the `config` crate and it will fail to pick up integers if using the standard deserialization routine from `serde`.
Luckily enough, we can specify a custom deserialization function.
Let's add a new dependency, `serde-aux` (`serde` auxiliary):

```
#! Cargo.toml
# [...]
[dependencies]
serde-aux = "3"
# [...]
```

and let's modify both `ApplicationSettings` and `DatabaseSettings`

```
//! src/configuration.rs
// [...]
use serde_aux::field_attributes::deserialize_number_from_string;
// [...]

#[derive(serde::Deserialize)]
pub struct ApplicationSettings {
    #[serde(deserialize_with = "deserialize_number_from_string")]
    pub port: u16,
    // [...]
}

#[derive(serde::Deserialize)]
pub struct DatabaseSettings {
    #[serde(deserialize_with = "deserialize_number_from_string")]
    pub port: u16,
    // [...]
}

// [...]
```

5.4.4 Connecting To Digital Ocean's Postgres Instance

Let's have a look at the connection string of our database using DigitalOcean's dashboard (Components -> Database):

```
postgresql://newsletter:<PASSWORD>@<HOST>:<PORT>/newsletter?sslmode=require
```

Our current `DatabaseSettings` does not handle SSL mode - it was not relevant for local development, but it is more than desirable to have transport-level encryption for our client/database communication in production.
Before trying to add new functionality, let's *make room for it* by refactoring `DatabaseSettings`.
The current version looks like this:

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

#[derive(serde::Deserialize)]
pub struct DatabaseSettings {
    pub username: String,
    pub password: Secret<String>,
    #[serde(deserialize_with = "deserialize_number_from_string")]
    pub port: u16,
    pub host: String,
    pub database_name: String,
}

impl DatabaseSettings {
    pub fn connection_string(&self) -> Secret<String> {
        // [...]
    }

    pub fn connection_string_without_db(&self) -> Secret<String> {
        // [...]
    }
}
```

We will change its two methods to return a `PgConnectOptions` instead of a connection string: it will make it easier to manage all these moving parts.

```rust
//! src/configuration.rs
use sqlx::postgres::PgConnectOptions;
// [...]

impl DatabaseSettings {
    // Renamed from `connection_string_without_db`
    pub fn without_db(&self) -> PgConnectOptions {
        PgConnectOptions::new()
            .host(&self.host)
            .username(&self.username)
            .password(&self.password.expose_secret())
            .port(self.port)
    }

    // Renamed from `connection_string`
    pub fn with_db(&self) -> PgConnectOptions {
        self.without_db().database(&self.database_name)
    }
}
```

We'll also have to update `src/main.rs` and `tests/health_check.rs`:

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

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

    let connection_pool = PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        // `connect_lazy_with` instead of `connect_lazy`
        .connect_lazy_with(configuration.database.with_db());

    // [...]
}
```

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

pub async fn configure_database(config: &DatabaseSettings) -> PgPool {
    // Create database
    let mut connection = PgConnection::connect_with(&config.without_db())
        .await
        .expect("Failed to connect to Postgres");
    connection
        .execute(format!(r#"CREATE DATABASE "{}";"#, config.database_name).as_str())
        .await
        .expect("Failed to create database.");

    // Migrate database
    let connection_pool = PgPool::connect_with(config.with_db())
        .await
        .expect("Failed to connect to Postgres.");
    sqlx::migrate!("./migrations")
        .run(&connection_pool)
        .await
        .expect("Failed to migrate the database");

    connection_pool
}
```

Use `cargo test` to make sure everything is still working as expected.

Let's now add the `require_ssl` property we need to `DatabaseSettings`:

```rust
//! src/configuration.rs
use sqlx::postgres::PgSslMode;
// [...]

#[derive(serde::Deserialize)]
pub struct DatabaseSettings {
    // [...]
    // Determine if we demand the connection to be encrypted or not
    pub require_ssl: bool,
}

impl DatabaseSettings {
    pub fn without_db(&self) -> PgConnectOptions {
        let ssl_mode = if self.require_ssl {
            PgSslMode::Require
        } else {
            // Try an encrypted connection, fallback to unencrypted if it fails
            PgSslMode::Prefer
        };
        PgConnectOptions::new()
            .host(&self.host)
            .username(&self.username)
            .password(&self.password.expose_secret())
            .port(self.port)
            .ssl_mode(ssl_mode)
    }
    // [...]
}
```

We want `require_ssl` to be `false` when we run the application locally (and for our test suite), but `true` in our production environment.
Let's amend our configuration files accordingly:

```yaml
#! configuration/local.yaml
application:
  host: 127.0.0.1
database:
  # New entry!
  require_ssl: false
```

```yaml
#! configuration/production.yaml
application:
  host: 0.0.0.0
database:
  # New entry!
  require_ssl: true
```

We can take the opportunity - now that we are using `PgConnectOptions` - to tune `sqlx`'s instrumentation: lower their logs from `INFO` to `TRACE` level.
This will eliminate the noise we noticed in the previous chapter.

```rust
//! src/configuration.rs
use sqlx::ConnectOptions;
// [...]

impl DatabaseSettings {
    // [...]
    pub fn with_db(&self) -> PgConnectOptions {
        let mut options = self.without_db().database(&self.database_name);
        options.log_statements(tracing::log::LevelFilter::Trace);
        options
    }
}
```

5.4.5 Environment Variables In The App Spec

One last step: we need to amend our `spec.yaml` manifest to inject the environment variables we need.

```yaml
#! spec.yaml
name: zero2prod
region: fra
services:
  - name: zero2prod
    # [...]
    envs:
      - key: APP_DATABASE__USERNAME
        scope: RUN_TIME
        value: ${newsletter.USERNAME}
      - key: APP_DATABASE__PASSWORD
        scope: RUN_TIME
        value: ${newsletter.PASSWORD}
      - key: APP_DATABASE__HOST
        scope: RUN_TIME
        value: ${newsletter.HOSTNAME}
      - key: APP_DATABASE__PORT
        scope: RUN_TIME
        value: ${newsletter.PORT}
      - key: APP_DATABASE__DATABASE_NAME
        scope: RUN_TIME
        value: ${newsletter.DATABASE}
databases:
  - name: newsletter
    # [...]
```

The scope is set to `RUN_TIME` to distinguish between environment variables needed during our Docker build process and those needed when the Docker image is launched.
We are populating the values of the environment variables by interpolating what is exposed by the Digital Ocean's platform (e.g. `${newsletter.PORT}`) - refer to their documentation for more details.

5.4.6 One Last Push

Let's apply the new spec

```bash
# You can retrieve your app id using `doctl apps list`
doctl apps update YOUR-APP-ID --spec=spec.yaml
```

and push our change up to GitHub to trigger a new deployment.

We now need to migrate the database[6]:

```bash
DATABASE_URL=YOUR-DIGITAL-OCEAN-DB-CONNECTION-STRING sqlx migrate run
```

We are ready to go!
Let's fire off a `POST` request to `/subscriptions`:

```bash
curl --request POST \
    --data 'name=le%20guin&email=ursula_le_guin%40gmail.com' \
```

[6] You will have to temporarily disable `Trusted Sources` to run the migrations from your local machine.

```
    https://zero2prod-adqrw.ondigitalocean.app/subscriptions \
    --verbose
```

The server should respond with a `200 OK`.
Congrats, you have just deployed your first Rust application!
And Ursula Le Guin just subscribed to your email newsletter (allegedly)!

If you have come this far, I'd love to get a screenshot of your Digital Ocean's dashboard showing off that running application!
Email it over at `rust@lpalmieri.com` or share it on Twitter tagging the *Zero To Production In Rust* account, zero2prod.

Chapter 6

Reject Invalid Subscribers #1

Our newsletter API is live, hosted on a Cloud provider.
We have a basic set of instrumentation to troubleshoot issues that might arise.
There is an exposed endpoint (`POST /subscriptions`) to subscribe to our content.

We have come a long way!

But we have cut a few corners along the way: `POST /subscriptions` is fairly… permissive.
Our input validation is extremely limited: we just ensure that both the name and the email fields are provided, nothing else.

We can add a new integration test to probe our API with some "troublesome" inputs:

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

#[tokio::test]
async fn subscribe_returns_a_200_when_fields_are_present_but_empty() {
    // Arrange
    let app = spawn_app().await;
    let client = reqwest::Client::new();
    let test_cases = vec![
        ("name=&email=ursula_le_guin%40gmail.com", "empty name"),
        ("name=Ursula&email=", "empty email"),
        ("name=Ursula&email=definitely-not-an-email", "invalid email"),
    ];

    for (body, description) in test_cases {
        // Act
        let response = client
            .post(&format!("{}/subscriptions", &app.address))
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(body)
            .send()
            .await
            .expect("Failed to execute request.");

        // Assert
        assert_eq!(
            200,
            response.status().as_u16(),
            "The API did not return a 200 OK when the payload was {}.",
            description
        );
    }
}
```

The new test, unfortunately, passes.
Although all those payloads are clearly invalid, our API is gladly accepting them, returning a `200 OK`.
Those troublesome subscriber details end up straight in our database, ready to give us problems down the line when it is time to deliver a newsletter issue.

We are asking for two pieces of information when subscribing to our newsletter: a name and an email.
This chapter will focus on name validation: what should we look out for?

6.1 Requirements

6.1.1 Domain Constraints

It turns out that names are complicated[1].
Trying to nail down what makes a name *valid* is a fool's errand. Remember that we chose to collect a name to use it in the opening line of our emails - we do not need it to match the real identity of a person, whatever that means in their geography. It would be totally unnecessary to inflict the pain of incorrect or overly prescriptive validation on our users.

We could thus settle on simply requiring the name field to be non-empty (as in, it must contain at least a non-whitespace character).

6.1.2 Security Constraints

Unfortunately, not all people on the Internet are good people.
Given enough time, especially if our newsletter picks up traction and becomes successful, we are bound to capture the attention of malicious visitors.
Forms and user inputs are a primary attack target - if they are not properly sanitised, they might allow an attacker to mess with our database (SQL injection), execute code on our servers, crash our service and other nasty stuff.
Thanks, but no thanks.

What is likely to happen in our case? What should we brace for in the wild range of possible attacks?[2]
We are building an email newsletter, which leads us to focus on:

- denial of service - e.g. trying to take our service down to prevent other people from signing up. A common threat for basically any online service;
- data theft - e.g. steal a huge list of email addresses;

[1] "Falsehoods programmers believe about names" by patio11 is a great starting point to deconstruct everything you believed to be true about peoples' names.

[2] In a more formalised context you would usually go through a threat-modelling exercise.

- phishing - e.g. use our service to send what looks like a legitimate email to a victim to trick them into clicking on some links or perform other actions.

Should we try to tackle all these threats in our validation logic?
Absolutely not!
But it is good practice to have a layered security approach[3]: by having mitigations to reduce the risk for those threats at multiple levels in our stack (e.g. input validation, parametrised queries to avoid SQL injection, escaping parametrised input in emails, etc.) we are less likely to be vulnerable should any of those checks fail us or be removed later down the line.

We should always keep in mind that software is a living artifact: holistic understanding of a system is the first victim of the passage of time.
You have the whole system in your head when writing it down for the first time, but the next developer touching it will not - at least not from the get-go. It is therefore possible for a load-bearing check in an obscure corner of the application to disappear (e.g. HTML escaping) leaving you exposed to a class of attacks (e.g. phishing).
Redundancy reduces risk.

Let's get to the point - what validation should we perform on names to improve our security posture given the class of threats we identified?
I suggest:

- Enforcing a maximum length. We are using `TEXT` as type for our email in Postgres, which is virtually unbounded - well, until disk storage starts to run out. Names come in all shapes and forms, but 256 characters should be enough for the greatest majority of our users[4] - if not, we will politely ask them to enter a nickname.
- Reject names containing troublesome characters. `/()"<>\{}` are fairly common in URLs, SQL queries and HTML fragments - not as much in names[5]. Forbidding them raises the complexity bar for SQL injection and phishing attempts.

6.2 First Implementation

Let's have a look at our request handler, as it stands right now:

```rust
//! src/routes/subscriptions.rs
use actix_web::{web, HttpResponse};
use chrono::Utc;
use sqlx::PgPool;
use uuid::Uuid;

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

#[tracing::instrument(
    name = "Adding a new subscriber",
    skip(form, pool),
```

[3] It is commonly referred to as *defense in depth*.
[4] Hubert B. Wolfe + 666 Sr would have been a victim of our maximum length check.
[5] Mandatory xkcd comic.

```
    fields(
        subscriber_email = %form.email,
        subscriber_name= %form.name
    )
)]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    match insert_subscriber(&pool, &form).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}

// [...]
```

Where should our new validation live?

A first sketch could look somewhat like this:

```
//! src/routes/subscriptions.rs

// An extension trait to provide the `graphemes` method
// on `String` and `&str`
use unicode_segmentation::UnicodeSegmentation;
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    if !is_valid_name(&form.name) {
        return HttpResponse::BadRequest().finish();
    }
    match insert_subscriber(&pool, &form).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}

/// Returns `true` if the input satisfies all our validation constraints
/// on subscriber names, `false` otherwise.
pub fn is_valid_name(s: &str) -> bool {
    // `.trim()` returns a view over the input `s` without trailing
    // whitespace-like characters.
    // `.is_empty` checks if the view contains any character.
    let is_empty_or_whitespace = s.trim().is_empty();

    // A grapheme is defined by the Unicode standard as a "user-perceived"
    // character: `å` is a single grapheme, but it is composed of two characters
    // (`a` and `̊`).
    //
    // `graphemes` returns an iterator over the graphemes in the input `s`.
    // `true` specifies that we want to use the extended grapheme definition set,
    // the recommended one.
    let is_too_long = s.graphemes(true).count() > 256;
    // Iterate over all characters in the input `s` to check if any of them matches
    // one of the characters in the forbidden array.
    let forbidden_characters = ['/', '(', ')', '"', '<', '>', '\\', '{', '}'];
    let contains_forbidden_characters = s.chars().any(|g| forbidden_characters.contains(&g));


    // Return `false` if any of our conditions have been violated
    !(is_empty_or_whitespace || is_too_long || contains_forbidden_characters)
}
```

To compile the new function successfully we will have to add the `unicode-segmentation` crate to our dependencies:

```
#! Cargo.toml
# [...]
[dependencies]
unicode-segmentation = "1"
# [...]
```

While it *looks like* a perfectly fine solution (assuming we add a bunch of tests), functions like `is_valid_name` give us a false sense of safety.

6.3 Validation Is A Leaky Cauldron

Let's shift our attention to `insert_subscriber`.
Let's imagine, for a second, that it requires `form.name` to be non-empty otherwise something horrible is going to happen (e.g. a panic!).

Can `insert_subscriber` safely assume that `form.name` will be non-empty?
Just by looking at its *type*, it cannot: `form.name` is a `String`. There is no guarantee about its content.
If you were to look at our program in its entirety you might say: we are checking that it is non-empty at the edge, in the request handler, therefore we can safely assume that `form.name` will be non-empty every time `insert_subscriber` is invoked.

But we had to shift from a *local* approach (let's look at this function's parameters) to a *global* approach (let's scan the whole codebase) to make such a claim.
And while it might be feasible for a small project such as ours, examining all the calling sites of a function (`insert_subscriber`) to ensure that a certain validation step has been performed beforehand quickly becomes unfeasible on larger projects.

If we are to stick with `is_valid_name`, the only viable approach is validating *again* `form.name` inside `insert_subscriber` - and every other function that requires our name to be non-empty.
That is the only way we can actually make sure that our invariant is in place where we need it.

What happens if `insert_subscriber` becomes too big and we have to split it out in multiple sub-functions? If they need the invariant, each of those has to perform validation to be certain it holds.
As you can see, this approach does not scale.

The issue here is that `is_valid_name` is a *validation function*: it tells us that, at a certain point in the execution flow of our program, a set of conditions is verified.

But this information about the additional structure in our input data **is not stored anywhere**. It is immediately lost.
Other parts of our program cannot reuse it effectively - they are forced to perform another point-in-time check leading to a crowded codebase with noisy (and wasteful) input checks at every step.

What we need is a *parsing function* - a routine that accepts unstructured input and, if a set of conditions holds, returns us a **more structured output**, an output that *structurally* guarantees that the invariants we care about hold from that point onwards.
How?

Using types!

6.4 Type-Driven Development

Let's add a new module to our project, `domain`, and define a new struct inside it, `SubscriberName`:

```
//! src/lib.rs
pub mod configuration;
// New module!
pub mod domain;
pub mod routes;
pub mod startup;
pub mod telemetry;
```

```
//! src/domain.rs

pub struct SubscriberName(String);
```

`SubscriberName` is a *tuple struct* - a new type, with a single (unnamed) field of type `String`.

`SubscriberName` is a proper new type, not just an alias - it does not inherit any of the methods available on `String` and trying to assign a `String` to a variable of type `SubscriberName` will trigger a compiler error - e.g.:

```
let name: SubscriberName = "A string".to_string();
```

```
error[E0308]: mismatched types
   |     let name: SubscriberName = "A string".to_string();
   |               --------------   ^^^^^^^^^^^^^^^^^^^^^^
   |               |                expected struct `SubscriberName`,
   |               |                found struct `std::string::String`
   |               |
   |               expected due to this
```

The inner field of `SubscriberName`, according to our current definition, is private: it can only be accessed from code within our `domain` module according to Rust's visibility rules.
As always, trust but verify: what happens if we try to build a `SubscriberName` in our `subscribe` request handler?

```
//! src/routes/subscriptions.rs
/// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let subscriber_name = crate::domain::SubscriberName(form.name.clone());
    /// [...]
}
```

The compiler complains with

```
error[E0603]: tuple struct constructor `SubscriberName` is private
  --> src/routes/subscriptions.rs:25:42
   |
25 |     let subscriber_name = crate::domain::SubscriberName(form.name.clone());
   |                                          ^^^^^^^^^^^^^^
   |                                          private tuple struct constructor
   |
  ::: src/domain.rs:1:27
   |
1  | pub struct SubscriberName(String);
   |                           ------ a constructor is private if
   |                                  any of the fields is private
```

It is therefore **impossible** (as it stands now) to build a `SubscriberName` instance outside of our `domain` module.
Let's add a new method to `SubscriberName`:

```
//! src/domain.rs
use unicode_segmentation::UnicodeSegmentation;

pub struct SubscriberName(String);

impl SubscriberName {
    /// Returns an instance of `SubscriberName` if the input satisfies all
    /// our validation constraints on subscriber names.
    /// It panics otherwise.
    pub fn parse(s: String) -> SubscriberName {
        // `.trim()` returns a view over the input `s` without trailing
        // whitespace-like characters.
        // `.is_empty` checks if the view contains any character.
        let is_empty_or_whitespace = s.trim().is_empty();

        // A grapheme is defined by the Unicode standard as a "user-perceived"
        // character: `å` is a single grapheme, but it is composed of two characters
        // (`a` and `̊`).
        //
        // `graphemes` returns an iterator over the graphemes in the input `s`.
        // `true` specifies that we want to use the extended grapheme definition set,
        // the recommended one.
        let is_too_long = s.graphemes(true).count() > 256;

        // Iterate over all characters in the input `s` to check if any of them matches
        // one of the characters in the forbidden array.
        let forbidden_characters = ['/', '(', ')', '"', '<', '>', '\\', '{', '}'];
        let contains_forbidden_characters = s.chars().any(|g| forbidden_characters.contains(&g));

        if is_empty_or_whitespace || is_too_long || contains_forbidden_characters {
            panic!("{} is not a valid subscriber name.", s)
        } else {
            Self(s)
        }
    }
}
```

Yes, you are right - that is a shameless copy-paste of what we had in `is_valid_name`.

There is one key difference though: the return type.
While `is_valid_name` gave us back a boolean, the `parse` method returns a `SubscriberName` if all checks are successful.

There is more!
`parse` is the only way to build an instance of `SubscriberName` outside of the `domain` module - we checked this was the case a few paragraphs ago.
We can therefore assert that *any* instance of `SubscriberName` will satisfy all our validation constraints.
We have made it **impossible** for an instance of `SubscriberName` to violate those constraints.

Let's define a new struct, `NewSubscriber`:

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

pub struct NewSubscriber {
    pub email: String,
    pub name: SubscriberName,
}

pub struct SubscriberName(String);

// [...]
```

What happens if we change `insert_subscriber` to accept an argument of type `NewSubscriber` instead of `FormData`?

```
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    // [...]
}
```

With the new signature we can be **sure** that `new_subscriber.name` is non-empty - it is **impossible** to call `insert_subscriber` passing an empty subscriber name.
And we can draw this conclusion just by looking up the definition of the types of the function arguments - we can once again make a *local* judgement, no need to go and check all the calling sites of our function.

Take a second to appreciate what just happened: we started with a set of requirements (all subscriber names must verify some constraints), we identified a potential pitfall (we might forget to validate the input before calling `insert_subscriber`) and **we leveraged Rust's type system to eliminate the pitfall, entirely**.
We made an incorrect usage pattern unrepresentable, by construction - it will not compile.

This technique is known as *type-driven development*[6].
Type-driven development is a powerful approach to encode the constraints of a domain we are trying to model inside the type system, leaning on the compiler to make sure they are enforced. The more expressive the type system of our programming language is, the tighter we can constrain our code to only be able to represent states that are valid in the domain we are working in.

Rust has not invented type-driven development - it has been around for a while, especially in the functional programming communities (Haskell, F#, OCaml, etc.). Rust "just" provides you with a type-system that is expressive enough to leverage many of the design patterns that have been pioneered in those languages in the past decades. The particular pattern we have just shown is often referred to as the "new-type pattern" in the Rust community.

We will be touching upon type-driven development as we progress in our implementation, but I strongly invite you to check out some of the resources mentioned in the footnotes of this chapter: they are treasure chests for any developer.

6.5 Ownership Meets Invariants

We changed `insert_subscriber`'s signature, but we have not amended the body to match the new requirements - let's do it now.

```rust
//! src/routes/subscriptions.rs
use crate::domain::{NewSubscriber, SubscriberName};
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    // `web::Form` is a wrapper around `FormData`
    // `form.0` gives us access to the underlying `FormData`
    let new_subscriber = NewSubscriber {
        email: form.0.email,
        name: SubscriberName::parse(form.0.name),
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}

#[tracing::instrument(
    name = "Saving new subscriber details in the database",
    skip(new_subscriber, pool)
)]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
```

[6] "Parse, don't validate" by Alexis King is a great starting point on type-driven development. "Domain Modelling Made Functional" by Scott Wlaschin is the perfect book to go deeper, with a specific focus around domain modelling - if a book looks like too much material, definitely check out Scott's talk.

```
    sqlx::query!(
        r#"
    INSERT INTO subscriptions (id, email, name, subscribed_at)
    VALUES ($1, $2, $3, $4)
        "#,
        Uuid::new_v4(),
        new_subscriber.email,
        new_subscriber.name,
        Utc::now()
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

Close enough - `cargo check` fails with:

```
error[E0308]: mismatched types
  --> src/routes/subscriptions.rs:50:9
   |
50 |         new_subscriber.name,
   |         ^^^^^^^^^^^^^^ expected `&str`,
   |         found struct `SubscriberName`
```

We have an issue here: we do not have any way to actually access the `String` value encapsulated inside `SubscriberName`!
We could change `SubscriberName`'s definition from `SubscriberName(String)` to `SubscriberName(pub String)`, but we would lose all the nice guarantees we spent the last two sections talking about:

- other developers would be allowed to bypass `parse` and build a `SubscriberName` with an arbitrary string

```
let liar = SubscriberName("".to_string());
```

- other developers might still choose to build a `SubscriberName` using `parse` but they would then have the option to mutate the inner value later to something that does not satisfy anymore the constraints we care about

```
let mut started_well = SubscriberName::parse("A valid name".to_string());
started_well.0 = "".to_string();
```

We can do better - this is the perfect place to take advantage of Rust's ownership system!
Given a field in a struct we can choose to:

- expose it by value, consuming the struct itself:

```
impl SubscriberName {
    pub fn inner(self) -> String {
        // The caller gets the inner string,
        // but they do not have a SubscriberName anymore!
        // That's because `inner` takes `self` by value,
```

```rust
        // consuming it according to move semantics
        self.0
    }
}
```

- expose a mutable reference:

```rust
impl SubscriberName {
    pub fn inner_mut(&mut self) -> &mut str {
        // The caller gets a mutable reference to the inner string.
        // This allows them to perform *arbitrary* changes to
        // value itself, potentially breaking our invariants!
        &mut self.0
    }
}
```

- expose a shared reference:

```rust
impl SubscriberName {
    pub fn inner_ref(&self) -> &str {
        // The caller gets a shared reference to the inner string.
        // This gives the caller **read-only** access,
        // they have no way to compromise our invariants!
        &self.0
    }
}
```

`inner_mut` is not what we are looking for here - the loss of control on our invariants would be equivalent to using `SubscriberName(pub String)`.
Both `inner` and `inner_ref` would be suitable, but `inner_ref` communicates better our intent: give the caller a chance to read the value without the power to mutate it.

Let's add `inner_ref` to `SubscriberName` - we can then amend `insert_subscriber` to use it:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
    INSERT INTO subscriptions (id, email, name, subscribed_at)
    VALUES ($1, $2, $3, $4)
        "#,
        Uuid::new_v4(),
        new_subscriber.email,
        // Using `inner_ref`!
        new_subscriber.name.inner_ref(),
        Utc::now()
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

Boom, it compiles!

6.5.1 AsRef

While our `inner_ref` method gets the job done, I am obliged to point out that Rust's standard library exposes a trait that is designed **exactly** for this type of usage - `AsRef`.

The definition is quite concise:

```rust
pub trait AsRef<T: ?Sized> {
    /// Performs the conversion.
    fn as_ref(&self) -> &T;
}
```

When should you implement `AsRef<T>` for a type?
When the type is *similar enough* to `T` that we can use a `&self` to get a reference to `T` itself!

Does it sound too abstract? Check out the signature of `inner_ref` again: that is basically `AsRef<str>` for `SubscriberName`!

`AsRef` can be used to improve ergonomics - let's consider a function with this signature:

```rust
pub fn do_something_with_a_string_slice(s: &str) {
    // [...]
}
```

To invoke it with our `SubscriberName` we would have to first call `inner_ref` and then call `do_something_with_a_string_slice`:

```rust
let name = SubscriberName::parse("A valid name".to_string());
do_something_with_a_string_slice(name.inner_ref())
```

Nothing too complicated, but it might take you some time to figure out *if* `SubscriberName` can give you a `&str` as well as *how*, especially if the type comes from a third-party library.
We can make the experience more seamless by changing `do_something_with_a_string_slice`'s signature:

```rust
// We are constraining T to implement the AsRef<str> trait
// using a trait bound - `T: AsRef<str>`
pub fn do_something_with_a_string_slice<T: AsRef<str>>(s: T) {
    let s = s.as_ref();
    // [...]
}
```

We can now write

```rust
let name = SubscriberName::parse("A valid name".to_string());
do_something_with_a_string_slice(name)
```

and it will compile straight-away (assuming `SubscriberName` implements `AsRef<str>`).

This pattern is used quite extensively, for example, in the filesystem module in Rust's standard library - `std::fs`. Functions like `create_dir` take an argument of type `P` constrained to implement `AsRef<Path>` instead of forcing the user to understand how to convert a `String` into a `Path`. Or how to convert a `PathBuf` into `Path`. Or an `OsString`. Or... you got the gist.

There are other little conversion traits like `AsRef` in that standard library - they provide a shared interface for the whole ecosystem to standardise around. Implementing them for your types suddenly unlocks a great deal of functionality exposed via generic types in the crates already available in the wild.
We will cover some of the other conversion trait later down the line (e.g. `From`/`Into`, `TryFrom`/`TryInto`).

Let's remove `inner_ref` and implement `AsRef<str>` for `SubscriberName`:

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

impl AsRef<str> for SubscriberName {
    fn as_ref(&self) -> &str {
        &self.0
    }
}
```

We also need to change `insert_subscriber`:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
    INSERT INTO subscriptions (id, email, name, subscribed_at)
    VALUES ($1, $2, $3, $4)
        "#,
        Uuid::new_v4(),
        new_subscriber.email,
        // Using `as_ref` now!
        new_subscriber.name.as_ref(),
        Utc::now()
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

The project compiles...

6.6 Panics

…but our tests are not green:

```
thread 'actix-rt:worker:0' panicked at
' is not a valid subscriber name.', src/domain.rs:39:13

[…]

---- subscribe_returns_a_200_when_fields_are_present_but_empty stdout ----
thread 'subscribe_returns_a_200_when_fields_are_present_but_empty' panicked at
'Failed to execute request.:
  reqwest::Error {
    kind: Request,
    url: Url {
      scheme: "http",
      host: Some(Ipv4(127.0.0.1)),
      port: Some(40681),
      path: "/subscriptions",
      query: None,
      fragment: None
    },
    source: hyper::Error(IncompleteMessage)
  }',
tests/health_check.rs:164:14
Panic in Arbiter thread.
```

On the bright side: we are not returning a `200 OK` anymore for empty names.
On the not-so-bright side: our API is terminating the request processing abruptly, causing the client to observe an `IncompleteMessage` error. Not very graceful.

Let's change the test to reflect our new expectations: we'd like to see a `400 Bad Request` response when the payload contains invalid data.

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

#[tokio::test]
// Renamed!
async fn subscribe_returns_a_400_when_fields_are_present_but_invalid() {
    // […]

    assert_eq!(
        // Not 200 anymore!
        400,
        response.status().as_u16(),
        "The API did not return a 400 Bad Request when the payload was {}.",
        description
    );

    // […]
}
```

Now, let's look at the root cause - we chose to panic when validation checks in `SubscriberName::parse` fail:

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

impl SubscriberName {
    pub fn parse(s: String) -> SubscriberName {
        // [...]

        if is_empty_or_whitespace || is_too_long || contains_forbidden_characters {
            panic!("{} is not a valid subscriber name.", s)
        } else {
            Self(s)
        }
    }
}
```

Panics in Rust are used to deal with **unrecoverable** errors: failure modes that were not expected or that we have no way to meaningfully recover from. Examples might include the host machine running out of memory or a full disk.
Rust's panics are **not** equivalent to exceptions in languages such as Python, C# or Java. Although Rust provides a few utilities to catch (some) panics, it is most definitely not the recommended approach and should be used sparingly.

`burntsushi` put it down quite neatly in a Reddit thread a few years ago:

> [...] If your Rust application panics in response to any user input, then the following should be true: your application has a bug, whether it be in a library or in the primary application code.

Adopting this viewpoint we can understand what is happening: when our request handler panics `actix-web` assumes that something horrible happened and immediately drops the worker that was dealing with that panicking request.[7]

If panics are not the way to go, what should we use to handle **recoverable** errors?

6.7 Error As Values - `Result`

Rust's primary error handling mechanism is built on top of the `Result` type:

```rust
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` is used as the return type for fallible operations: if the operation succeeds, `Ok(T)` is returned; if it fails, you get `Err(E)`.
We have actually already used `Result`, although we did not stop to discuss its nuances at the time.
Let's look again at the signature of `insert_subscriber`:

[7] A panic in a request handler does not crash the **whole** application. `actix-web` spins up multiple workers to deal with incoming requests and it is resilient to one or more of them crashing: it will just spawn new ones to replace the ones that failed.

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    // [...]
}
```

It tells us that inserting a subscriber in the database is a fallible operation - if all goes as planned, we don't get anything back (`()` - the unit type), if something is amiss we will instead receive a `sqlx::Error` with details about what went wrong (e.g. a connection issue).

Errors as values, combined with Rust's enums, are awesome building blocks for a robust error handling story.
If you are coming from a language with exception-based error handling, this is likely to be a game changer[8]: everything we need to know about the failure modes of a function is in its signature.

You will not have to dig in the documentation of your dependencies to understand what exceptions a certain function might throw (assuming it is documented in the first place!).
You will not be surprised at runtime by yet another undocumented exception type.
You will not have to insert a catch-all statement "just in case".

We will cover the basics here and leave the finer details (`Error` trait) to the next chapter.

6.7.1 Converting `parse` To Return `Result`

Let's refactor our `SubscriberName::parse` to return a `Result` instead of panicking on invalid inputs.
We will start by changing the signature, without touching the body:

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

impl SubscriberName {
    pub fn parse(s: String) -> Result<SubscriberName, ???> {
        // [...]
    }
}
```

What type should we use as `Err` variant for our `Result`?
The simplest option is a `String` - we just return an error message on failure.

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

impl SubscriberName {
    pub fn parse(s: String) -> Result<SubscriberName, String> {
        // [...]
    }
}
```

[8] Checked exceptions in Java are the only example I am aware of in mainstream languages using exceptions that comes close enough to the compile-time safety provided by `Result`.

Running `cargo check` surfaces two errors from the compiler:

```
error[E0308]: mismatched types
  --> src/routes/subscriptions.rs:27:15
   |
27 |         name: SubscriberName::parse(form.0.name),
   |               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |               expected struct `SubscriberName`,
   |               found enum `Result`

error[E0308]: mismatched types
  --> src/domain.rs:41:13
   |
14 |     pub fn parse(s: String) -> Result<SubscriberName, String> {
   |                                ------------------------------
   |                                expected `Result<SubscriberName, String>`
   |                                because of return type
...
41 |             Self(s)
   |             ^^^^^^^
   |             |
   |             expected enum `Result`, found struct `SubscriberName`
   |             help: try using a variant of the expected enum: `Ok(Self(s))`
   |
   = note: expected enum `Result<SubscriberName, String>`
              found struct `SubscriberName`
```

Let's focus on the second error: we cannot return a bare instance of `SubscriberName` at the end of `parse` - we need to choose one of the two `Result` variants.
The compiler understands the issue and suggests the right edit: use `Ok(Self(s))` instead of `Self(s)`. Let's follow its advice:

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

impl SubscriberName {
    pub fn parse(s: String) -> Result<SubscriberName, String> {
        // [...]

        if is_empty_or_whitespace || is_too_long || contains_forbidden_characters {
            panic!("{} is not a valid subscriber name.", s)
        } else {
            Ok(Self(s))
        }
    }
}
```

`cargo check` should now return a single error:

```
error[E0308]: mismatched types
  --> src/routes/subscriptions.rs:27:15
   |
27 |         name: SubscriberName::parse(form.0.name),
   |               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |               expected struct `SubscriberName`,
   |               found enum `Result`
```

It is complaining about our invocation of the `parse` method in `subscribe`: when `parse` returned a `SubscriberName` it was perfectly fine to assign its output directly to `Subscriber.name`.
We are returning a `Result` now - Rust's type system **forces us** to deal with the unhappy path. We cannot just pretend it won't happen.
Let's avoid covering too much ground at once though - for the time being we will just panic if validation fails in order to get the project to compile again as quickly as possible:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let new_subscriber = NewSubscriber {
        email: form.0.email,
        // Notice the usage of `expect` to specify a meaningful panic message
        name: SubscriberName::parse(form.0.name).expect("Name validation failed."),
    };
    // [...]
}
```

`cargo check` should be happy now.
Time to work on tests!

6.8 Insightful Assertion Errors: `claim`

Most of our assertions will be along the lines of `assert!(result.is_ok())` or `assert!(result.is_err())`.
The error messages returned by `cargo test` on failure when using these assertions are quite poor. How poor? Let's run a quick experiment!

If you run `cargo test` on this dummy test

```rust
#[test]
fn dummy_fail() {
    let result: Result<&str, &str> = Err("The app crashed due to an IO error");
    assert!(result.is_ok());
}
```

you will get

```
---- dummy_fail stdout ----
thread 'dummy_fail' panicked at 'assertion failed: result.is_ok()'
```

We do not get any detail concerning the error itself - it makes for a somewhat painful debugging experience.
We will be using the `claim` crate to get more informative error messages:

```toml
#! Cargo.toml
# [...]
[dev-dependencies]
claim = "0.5"
# [...]
```

`claim` provides a fairly comprehensive range of assertions to work with common Rust types - in particular `Option` and `Result`.
If we rewrite our `dummy_fail` test to use `claim`

```rust
#[test]
fn dummy_fail() {
    let result: Result<&str, &str> = Err("The app crashed due to an IO error");
    claim::assert_ok!(result);
}
```

we get

```
---- dummy_fail stdout ----
thread 'dummy_fail' panicked at 'assertion failed, expected Ok(..),
  got Err("The app crashed due to an IO error")'
```

Much better.

6.9 Unit Tests

We are all geared up - let’s add some unit tests to the `domain` module to make sure all the code we wrote behaves as expected.

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

#[cfg(test)]
mod tests {
    use crate::domain::SubscriberName;
    use claim::{assert_err, assert_ok};

    #[test]
    fn a_256_grapheme_long_name_is_valid() {
        let name = "ë".repeat(256);
        assert_ok!(SubscriberName::parse(name));
    }

    #[test]
    fn a_name_longer_than_256_graphemes_is_rejected() {
        let name = "a".repeat(257);
        assert_err!(SubscriberName::parse(name));
    }

    #[test]
    fn whitespace_only_names_are_rejected() {
        let name = " ".to_string();
        assert_err!(SubscriberName::parse(name));
    }

    #[test]
    fn empty_string_is_rejected() {
        let name = "".to_string();
        assert_err!(SubscriberName::parse(name));
    }

    #[test]
    fn names_containing_an_invalid_character_are_rejected() {
        for name in &['/', '(', ')', '"', '<', '>', '\\', '{', '}'] {
            let name = name.to_string();
            assert_err!(SubscriberName::parse(name));
        }
    }

    #[test]
    fn a_valid_name_is_parsed_successfully() {
        let name = "Ursula Le Guin".to_string();
        assert_ok!(SubscriberName::parse(name));
    }
}
```

Unfortunately, it does not compile - `cargo` highlights all our usages of `assert_ok/assert_err` with

```
66 |         assert_err!(SubscriberName::parse(name));
   |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |         `SubscriberName` cannot be formatted using `{:?}`
   |
   = help: the trait `std::fmt::Debug` is not implemented for `SubscriberName`
   = note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
   = note: required by `std::fmt::Debug::fmt`
```

`claim` needs our type to implement the `Debug` trait to provide those nice error messages. Let's add a `#[derive(Debug)]` attribute on top of `SubscriberName`:

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

#[derive(Debug)]
pub struct SubscriberName(String);
```

The compiler should be happy now. What about tests?

```
cargo test
```

```
failures:
    domain::tests::a_name_longer_than_256_graphemes_is_rejected
    domain::tests::empty_string_is_rejected
    domain::tests::names_containing_an_invalid_character_are_rejected
    domain::tests::whitespace_only_names_are_rejected

test result: FAILED. 2 passed; 4 failed; 0 ignored; 0 measured; 0 filtered out
```

All our unhappy-path tests are failing because we are still panicking if our validation constraints are not satisfied - let's change it:

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

impl SubscriberName {
    pub fn parse(s: String) -> Result<SubscriberName, String> {
        // [...]

        if is_empty_or_whitespace || is_too_long || contains_forbidden_characters {
```

```rust
            // Replacing `panic!` with `Err(...)`
            Err(format!("{} is not a valid subscriber name.", s))
        } else {
            Ok(Self(s))
        }
    }
}
```

All our domain unit tests are now passing - let's finally address the failing integration test we wrote at the beginning of the chapter.

6.10 Handling A `Result`

`SubscriberName::parse` is now returning a `Result`, but `subscribe` is calling `expect` on it, therefore panicking if an `Err` variant is returned.
The behaviour of the application, as a whole, has not changed at all.

How do we change `subscribe` to return a `400 Bad Request` on validation errors? We can have a look at what we are already doing for our call to `insert_subscriber`!

6.10.1 `match`

How do we handle the possibility of a failure on the caller side?

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    // [...]
}
```

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    // [...]
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

`insert_subscriber` returns a `Result<(), sqlx::Error>` while `subscribe` speaks the language of a REST API - its output must be of type `HttpResponse`. To return a `HttpResponse` to the caller in the error case we need to convert `sqlx::Error` into a representation that makes sense within the technical domain of a REST API - in our case, a `500 Internal Server Error`.

That's where a `match` comes in handy: we tell the compiler what to do in both scenarios, `Ok` and

`Err`.

6.10.2 The `?` Operator

Speaking of error handling, let's look again at `insert_subscriber`:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber(/* */) -> Result<(), sqlx::Error> {
    sqlx::query!(/* */)
        .execute(pool)
        .await
        .map_err(|e| {
            tracing::error!("Failed to execute query: {:?}", e);
            e
        })?;
    Ok(())
}
```

Have you noticed that `?`, before `Ok(())`?

It is the question mark operator, `?`.
`?` was introduced in Rust `1.13` - it is syntactic sugar.
It reduces the amount of visual noise when you are working with fallible functions and you want to "bubble up" failures (e.g. similar enough to re-throwing a caught exception).

The `?` in this block

```rust
insert_subscriber(&pool, &new_subscriber)
.await
.map_err(|_| HttpResponse::InternalServerError().finish())?;
```

is equivalent to this control flow block

```rust
if let Err(error) = insert_subscriber(&pool, &new_subscriber)
    .await
    .map_err(|_| HttpResponse::InternalServerError().finish())
{
    return Err(error);
}
```

It allows us to return early when something fails using a single character instead of a multi-line block.

Given that `?` triggers an early return using an `Err` variant, it can only be used within a function that returns a `Result`. `subscribe` does not qualify (yet).

6.10.3 400 Bad Request

Let's handle now the error returned by `SubscriberName::parse`:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        // Return early if the name is invalid, with a 400
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber {
        email: form.0.email,
        name,
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

`cargo test` is not green yet, but we are getting a different error:

```
--- subscribe_returns_a_400_when_fields_are_present_but_invalid stdout ----
thread 'subscribe_returns_a_400_when_fields_are_present_but_invalid'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `200`:
 The API did not return a 400 Bad Request when the payload was empty email.',
tests/health_check.rs:167:9
```

The test case using an empty name is now passing, but we are failing to return a `400 Bad Request` when an empty email is provided.
Not unexpected - we have not implemented any kind of email validation yet!

6.11 The Email Format

We are all intuitively familiar with the *common* structure of an email address - `XXX@YYY.ZZZ` - but the subject quickly gets more complicated if you desire to be rigorous and avoid bouncing email addresses that are actually valid.

How do we establish if an email address is "valid"?
There are a few Request For Comments (RFC) by the Internet Engineering Task Force (IETF) outlining the expected structure of an email address - RFC 6854, RFC 5322, RFC 2822. We would have to read them, digest the material and then come up with an `is_valid_email` function that matches the specification.
Unless you have a keen interest in understanding the subtle nuances of the email address format, I would suggest you to take a step back: it is quite messy. So messy that even the HTML specification is *willfully* non-compliant with the RFCs we just linked.

Our best shot is to look for an existing library that has stared long and hard at the problem to provide us with a plug-and-play solution. Luckily enough, there is at least one in the Rust ecosystem - the `validator` crate![9]

[9] The `validator` crate follows the HTML specification when it comes to email validation. You can check its source code if you are curious to see how it's implemented.

6.12 The SubscriberEmail Type

We will follow the same strategy we used for name validation - encode our invariant ("this string represents a valid email") in a new `SubscriberEmail` type.

6.12.1 Breaking The Domain Sub-Module

Before we get started though, let's make some space - let's break our `domain` sub-module (`domain.rs`) into multiple smaller files, one for each type, similarly to what we did for routes back in Chapter 3. Our current folder structure (under `src`) is:

```
src/
  routes/
    [ ... ]
  domain.rs
  [ ... ]
```

We want to have

```
src/
  routes/
    [ ... ]
  domain/
    mod.rs
    subscriber_name.rs
    subscriber_email.rs
    new_subscriber.rs
  [ ... ]
```

Unit tests should be in the same file of the type they refer to. We will end up with:

```rust
//! src/domain/mod.rs

mod subscriber_name;
mod subscriber_email;
mod new_subscriber;

pub use subscriber_name::SubscriberName;
pub use new_subscriber::NewSubscriber;
```

```rust
//! src/domain/subscriber_name.rs

use unicode_segmentation::UnicodeSegmentation;

#[derive(Debug)]
pub struct SubscriberName(String);

impl SubscriberName {
    // [...]
}

impl AsRef<str> for SubscriberName {
    // [...]
}
```

```rust
#[cfg(test)]
mod tests {
    // [...]
}
```

```rust
//! src/domain/subscriber_email.rs

// Still empty, ready for us to get started!
```

```rust
//! src/domain/new_subscriber.rs

use crate::domain::subscriber_name::SubscriberName;

pub struct NewSubscriber {
    pub email: String,
    pub name: SubscriberName,
}
```

No changes should be required to other files in our project - the API of our module has not changed thanks to our `pub use` statements in `mod.rs`.

6.12.2 Skeleton Of A New Type

Let's add a barebone `SubscriberEmail` type: no validation, just a wrapper around a `String` and a convenient `AsRef` implementation:

```rust
//! src/domain/subscriber_email.rs

#[derive(Debug)]
pub struct SubscriberEmail(String);

impl SubscriberEmail {
    pub fn parse(s: String) -> Result<SubscriberEmail, String> {
        // TODO: add validation!
        Ok(Self(s))
    }
}

impl AsRef<str> for SubscriberEmail {
    fn as_ref(&self) -> &str {
        &self.0
    }
}
```

```rust
//! src/domain/mod.rs

mod new_subscriber;
mod subscriber_email;
mod subscriber_name;

pub use new_subscriber::NewSubscriber;
pub use subscriber_email::SubscriberEmail;
pub use subscriber_name::SubscriberName;
```

We start with tests this time: let's come up with a few examples of invalid emails that should be rejected.

```rust
//! src/domain/subscriber_email.rs

#[derive(Debug)]
pub struct SubscriberEmail(String);

// [...]

#[cfg(test)]
mod tests {
    use super::SubscriberEmail;
    use claim::assert_err;

    #[test]
    fn empty_string_is_rejected() {
        let email = "".to_string();
        assert_err!(SubscriberEmail::parse(email));
    }

    #[test]
    fn email_missing_at_symbol_is_rejected() {
        let email = "ursuladomain.com".to_string();
        assert_err!(SubscriberEmail::parse(email));
    }

    #[test]
    fn email_missing_subject_is_rejected() {
        let email = "@domain.com".to_string();
        assert_err!(SubscriberEmail::parse(email));
    }
}
```

Running `cargo test domain` confirms that all test cases are failing:

```
failures:
    domain::subscriber_email::tests::email_missing_at_symbol_is_rejected
    domain::subscriber_email::tests::email_missing_subject_is_rejected
    domain::subscriber_email::tests::empty_string_is_rejected

test result: FAILED. 6 passed; 3 failed; 0 ignored; 0 measured; 0 filtered out
```

Time to bring `validator` in:

```toml
#! Cargo.toml
# [...]
[dependencies]
validator = "0.14"
# [...]
```

Our `parse` method will just delegate all the heavy-lifting to `validator::validate_email`:

```rust
//! src/domain/subscriber_email.rs

use validator::validate_email;

#[derive(Debug)]
pub struct SubscriberEmail(String);
```

```rust
impl SubscriberEmail {
    pub fn parse(s: String) -> Result<SubscriberEmail, String> {
        if validate_email(&s) {
            Ok(Self(s))
        } else {
            Err(format!("{} is not a valid subscriber email.", s))
        }
    }
}

// [...]
```

As simple as that - all our tests are green now!
There is a caveat - all our tests cases are checking for *invalid* emails. We should also have at least one test checking that valid emails are going through.

We could hard-code a known valid email address in a test and check that it is parsed successfully - e.g. `ursula@domain.com`.
What value would we get from that test case though? It would only re-assure us that *a specific email address* is correctly parsed as valid.

6.13 Property-based Testing

We could use another approach to test our parsing logic: instead of verifying that a certain set of inputs is correctly parsed, we could build a random generator that produces valid values and check that our parser does not reject them.
In other words, we verify that our implementation displays a certain *property* - "No valid email address is rejected".

This approach is often referred to as *property-based testing*.

If we were working with time, for example, we could *repeatedly* sample three random integers

- `H`, between `0` and `23` (inclusive);
- `M`, between `0` and `59` (inclusive);
- `S`, between `0` and `59` (inclusive);

and verify that `H:M:S` is always correctly parsed.

Property-based testing significantly increases the range of inputs that we are validating, and therefore our confidence in the correctness of our code, but it does not *prove* that our parser is correct - it does not *exhaustively* explore the input space (except for tiny ones).

Let's see what property testing would look like for our `SubscriberEmail`.

6.13.1 How To Generate Random Test Data With `fake`

First and foremost, we need a random generator of valid emails.
We could write one, but this a great opportunity to introduce the `fake` crate.

`fake` provides generation logic for both primitive data types (integers, floats, strings) and higher-level objects (IP addresses, country codes, etc.) - in particular, emails! Let's add `fake` as a development dependency of our project:

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

[dev-dependencies]
# [...]
# We are not using fake >= 2.4 because it relies on rand 0.8
# which has been recently released and it is not yet used by
# quickcheck (solved in its upcoming 1.0 release!)
fake = "~2.3"
```

Let's use it in a new test:

```rust
//! src/domain/subscriber_email.rs

// [...]

#[cfg(test)]
mod tests {
    // We are importing the `SafeEmail` faker!
    // We also need the `Fake` trait to get access to the
    // `.fake` method on `SafeEmail`
    use fake::faker::internet::en::SafeEmail;
    use fake::Fake;
    // [...]

    #[test]
    fn valid_emails_are_parsed_successfully() {
        let email = SafeEmail().fake();
        claim::assert_ok!(SubscriberEmail::parse(email));
    }
}
```

Every time we run our test suite, `SafeEmail().fake()` generates a new random valid email which we then use to test our parsing logic.
This is already a major improvement compared to a hard-coded valid email, but we would have to run our test suite several times to catch an issue with an edge case. A fast-and-dirty solution would be to add a `for` loop to the test, but, once again, we can use this as an occasion to delve deeper and explore one of the available testing crates designed around property-based testing.

6.13.2 `quickcheck` Vs `proptest`

There are two mainstream options for property-based testing in the Rust ecosystem: `quickcheck` and `proptest`.
Their domains overlap, although each shines in its own niche - check their READMEs for all the nitty gritty details.

For our project we will go with `quickcheck` - it is fairly simple to get started with and it does not use too many macros, which makes for a pleasant IDE experience.

6.13.3 Getting Started With `quickcheck`

Let's have a look at one of their examples to get the gist of how it works:

```rust
/// The function we want to test.
fn reverse<T: Clone>(xs: &[T]) -> Vec<T> {
    let mut rev = vec!();
    for x in xs.iter() {
        rev.insert(0, x.clone())
    }
    rev
}

#[cfg(test)]
mod tests {
    #[quickcheck_macros::quickcheck]
    fn prop(xs: Vec<u32>) -> bool {
        /// A property that is always true, regardless
        /// of the vector we are applying the function to:
        /// reversing it twice should return the original input.
        xs == reverse(&reverse(&xs))
    }
}
```

`quickcheck` calls `prop` in a loop with a configurable number of iterations (100 by default): on every iteration, it generates a new `Vec<u32>` and checks that `prop` returned `true`.
If `prop` returns `false`, it tries to **shrink** the generated input to the smallest possible failing example (the shortest failing vector) to help us debug what went wrong.

In our case, we'd like to have something along these lines:

```rust
#[quickcheck_macros::quickcheck]
fn valid_emails_are_parsed_successfully(valid_email: String) -> bool {
    SubscriberEmail::parse(valid_email).is_ok()
}
```

Unfortunately, if we ask for a `String` type as input we are going to get all sorts of garbage which will fail validation.
How do we customise the generation routine?

6.13.4 Implementing The `Arbitrary` Trait

Let's go back to the previous example - how does `quickcheck` know how to generate a `Vec<u32>`?
Everything is built on top of `quickcheck`'s `Arbitrary` trait:

```rust
pub trait Arbitrary: Clone + Send + 'static {
    fn arbitrary<G: Gen>(g: &mut G) -> Self;

    fn shrink(&self) -> Box<dyn Iterator<Item = Self>> {
        empty_shrinker()
    }
}
```

We have two methods:

- `arbitrary`: given a source of randomness (`g`) it returns an instance of the type;
- `shrink`: it returns a sequence of progressively "smaller" instances of the type to help `quickcheck` find the smallest possible failure case.

`Vec<u32>` implements `Arbitrary`, therefore `quickcheck` knows how to generate random `u32` vectors.
We need to create our own type, let's call it `ValidEmailFixture`, and implement `Arbitrary` for it.
If you look at `Arbitrary`'s trait definition, you'll notice that shrinking is optional: there is a default implementation (using `empty_shrinker`) which results in `quickcheck` outputting the first failure encountered, without trying to make it any smaller or nicer. Therefore we only need to provide an implementation of `Arbitrary::arbitrary` for our `ValidEmailFixture`.

Let's add both `quickcheck` and `quickcheck-macros` as development dependencies:

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

[dev-dependencies]
# [...]
quickcheck = "0.9.2"
quickcheck_macros = "0.9.1"
```

Then

```rust
//! src/domain/subscriber_email.rs
// [...]

#[cfg(test)]
mod tests {
    // We have removed the `assert_ok` import.
    use claim::assert_err;
    // [...]

    // Both `Clone` and `Debug` are required by `quickcheck`
    #[derive(Debug, Clone)]
    struct ValidEmailFixture(pub String);

    impl quickcheck::Arbitrary for ValidEmailFixture {
        fn arbitrary<G: quickcheck::Gen>(g: &mut G) -> Self {
            let email = SafeEmail().fake_with_rng(g);
            Self(email)
        }
    }

    #[quickcheck_macros::quickcheck]
    fn valid_emails_are_parsed_successfully(valid_email: ValidEmailFixture) -> bool {
        SubscriberEmail::parse(valid_email.0).is_ok()
    }
}
```

This is an amazing example of the interoperability you gain by sharing key traits across the Rust ecosystem.
How do we get `fake` and `quickcheck` to play nicely together?

In `Arbitrary::arbitrary` we get `g` as input, an argument of type `G`.
`G` is constrained by a trait bound, `G: quickcheck::Gen`, therefore it must implement the `Gen` trait in `quickcheck`, where `Gen` stands for "generator".
How is `Gen` defined?

```
pub trait Gen: RngCore {
    fn size(&self) -> usize;
}
```

Anything that implements `Gen` must also implement the `RngCore` trait from `rand-core`.

Let's examine the `SafeEmail` faker: it implements the `Fake` trait.
`Fake` gives us a `fake` method, which we have already tried out, but it also exposes a `fake_with_rng` method, where "rng" stands for "random number generator".
What does `fake` accept as a valid random number generator?

```
pub trait Fake: Sized {
    //[...]

    fn fake_with_rng<U, R>(&self, rng: &mut R) -> U where
        R: Rng + ?Sized,
        Self: FakeBase<U>;
}
```

You read that right - any type that implements the `Rng` trait from `rand`, which is automatically implemented by all types implementing `RngCore`!
We can just pass `g` from `Arbitrary::arbitrary` as the random number generator for `fake_with_rng` and *everything just works*!

Maybe the maintainers of the two crates are aware of each other, maybe they aren't, but a community-sanctioned set of traits in `rand-core` gives us painless interoperability. Pretty sweet!

You can now run `cargo test domain` - it should come out green, re-assuring us that our email validation check is indeed not overly prescriptive.

If you want to see the random inputs that are being generated, add a `dbg!(&valid_email.0);` statement to the test and run `cargo test valid_emails -- --nocapture` - tens of valid emails should pop up in your terminal!

6.14 Payload Validation

If you run `cargo test`, without restricting the set of tests being run to `domain`, you will see that our integration test with invalid data is still red.

```
--- subscribe_returns_a_400_when_fields_are_present_but_invalid stdout ----
thread 'subscribe_returns_a_400_when_fields_are_present_but_invalid'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `200`:
 The API did not return a 400 Bad Request when the payload was empty email.',
tests/health_check.rs:167:9
```

Let's integrate our shiny `SubscriberEmail` into the application to benefit from its validation in our `/subscriptions` endpoint.
We need to start from `NewSubscriber`:

```
//! src/domain/new_subscriber.rs

use crate::domain::SubscriberName;
```

```rust
use crate::domain::SubscriberEmail;

pub struct NewSubscriber {
    // We are not using `String` anymore!
    pub email: SubscriberEmail,
    pub name: SubscriberName,
}
```

Hell should break loose if you try to compile the project now.
Let's start with the first error reported by `cargo check`:

```
error[E0308]: mismatched types
  --> src/routes/subscriptions.rs:28:16
   |
28 |         email: form.0.email,
   |                ^^^^^^^^^^^^
   |                expected struct `SubscriberEmail`,
   |                found struct `std::string::String`
```

It is referring to a line in our request handler, `subscribe`:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber {
        // We are trying to assign a string to a field of type SubscriberEmail!
        email: form.0.email,
        name,
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

We need to mimic what we are already doing for the `name` field: first we parse `form.0.email` then we assign the result (if successful) to `NewSubscriber.email`.

```rust
//! src/routes/subscriptions.rs

// We added `SubscriberEmail`!
use crate::domain::{NewSubscriber, SubscriberEmail, SubscriberName};
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let email = match SubscriberEmail::parse(form.0.email) {
        Ok(email) => email,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber { email, name };
    // [...]
}
```

Time to move to the second error:

```
error[E0308]: mismatched types
  --> src/routes/subscriptions.rs:50:9
   |
50 |         new_subscriber.email,
   |         ^^^^^^^^^^^^^^
   |         expected `&str`,
   |         found struct `SubscriberEmail`
```

This is in our `insert_subscriber` function, where we perform a SQL INSERT query to store the details of the new subscriber:

```rust
//! src/routes/subscriptions.rs

// [...]

#[tracing::instrument([...])]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
    INSERT INTO subscriptions (id, email, name, subscribed_at)
    VALUES ($1, $2, $3, $4)
            "#,
        Uuid::new_v4(),
        // It expects a `&str` but we are passing it
        // a `SubscriberEmail` value
        new_subscriber.email,
        new_subscriber.name.as_ref(),
        Utc::now()
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

The solution is right there, on the line below - we just need to borrow the inner field of

`SubscriberEmail` as a string slice using our implementation of `AsRef<str>`.

```rust
//! src/routes/subscriptions.rs

// [...]

#[tracing::instrument([...])]
pub async fn insert_subscriber(
    pool: &PgPool,
    new_subscriber: &NewSubscriber,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
    INSERT INTO subscriptions (id, email, name, subscribed_at)
    VALUES ($1, $2, $3, $4)
            "#,
        Uuid::new_v4(),
        // Using `as_ref` now!
        new_subscriber.email.as_ref(),
        new_subscriber.name.as_ref(),
        Utc::now()
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

That's it - it compiles now!
What about our integration test?

```
cargo test
```

```
running 4 tests
test subscribe_returns_a_400_when_data_is_missing ... ok
test health_check_works ... ok
test subscribe_returns_a_400_when_fields_are_present_but_invalid ... ok
test subscribe_returns_a_200_for_valid_form_data ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

All green! We made it!

6.14.1 Refactoring With `TryFrom`

Before we move on let's spend a few paragraphs to refactor the code we just wrote.
I am referring to our request handler, `subscribe`:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let name = match SubscriberName::parse(form.0.name) {
        Ok(name) => name,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let email = match SubscriberEmail::parse(form.0.email) {
        Ok(email) => email,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    let new_subscriber = NewSubscriber { email, name };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

We can extract the first two statements in a `parse_subscriber` function:

```rust
//! src/routes/subscriptions.rs
// [...]

pub fn parse_subscriber(form: FormData) -> Result<NewSubscriber, String> {
    let name = SubscriberName::parse(form.name)?;
    let email = SubscriberEmail::parse(form.email)?;
    Ok(NewSubscriber { email, name })
}

#[tracing::instrument([...])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let new_subscriber = match parse_subscriber(form.0) {
        Ok(subscriber) => subscriber,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

The refactoring gives us a clearer separation of concerns:

- `parse_subscriber` takes care of the conversion from our *wire format* (the url-decoded data collected from a HTML form) to our *domain model* (`NewSubscriber`);
- `subscribe` remains in charge of generating the HTTP response to the incoming HTTP request.

The Rust standard library provides a few traits to deal with conversions in its `std::convert` sub-module. That is where `AsRef` comes from!
Is there any trait there that captures what we are trying to do with `parse_subscriber`?
`AsRef` is not a good fit for what we are dealing with here: a *fallible* conversion between two types which **consumes** the input value.
We need to look at `TryFrom`:

```rust
pub trait TryFrom<T>: Sized {
    /// The type returned in the event of a conversion error.
    type Error;

    /// Performs the conversion.
    fn try_from(value: T) -> Result<Self, Self::Error>;
}
```

Replace `T` with `FormData`, `Self` with `NewSubscriber` and `Self::Error` with `String` - there you have it, the signature of our `parse_subscriber` function!
Let's try it out:

```rust
//! src/routes/subscriptions.rs
// No need to import the TryFrom trait, it is included
// in Rust's prelude since edition 2021!
// [...]

impl TryFrom<FormData> for NewSubscriber {
    type Error = String;

    fn try_from(value: FormData) -> Result<Self, Self::Error> {
        let name = SubscriberName::parse(value.name)?;
        let email = SubscriberEmail::parse(value.email)?;
        Ok(Self { email, name })
    }
}

#[tracing::instrument([...])]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let new_subscriber = match form.0.try_into() {
        Ok(form) => form,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

We implemented `TryFrom`, but we are calling `.try_into`? What is happening there?
There is another conversion trait in the standard library, called `TryInto`:

```rust
pub trait TryInto<T> {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

Its signature mirrors the one of `TryFrom` - the conversion just goes in the other direction!
If you provide a `TryFrom` implementation, your type automatically gets the corresponding `TryInto` implementation, for free.
`try_into` takes `self` as first argument, which allows us to do `form.0.try_into()` instead of going for `NewSubscriber::try_from(form.0)` - matter of taste, if you want.

Generally speaking, what do we gain by implementing `TryFrom/TryInto`?
Nothing shiny, no new functionality - we are "just" making our *intent* clearer.
We are spelling out "This is a type conversion!".

Why does it matter? It helps others!
When another developer with some Rust exposure jumps in our codebase they will immediately spot the conversion pattern because we are using a trait that they are already familiar with.

6.15 Summary

Validating that the email in the payload of `POST /subscriptions` complies with the expected format is good, but it is not enough.
We now have an email that is *syntactically* valid but we are still uncertain about its *existence*: does anybody actually use that email address? Is it reachable?
We have no idea and there is only one way to find out: sending an actual email.

Confirmation emails (and how to write a HTTP client!) will be the topic of the next chapter.

Chapter 7

Reject Invalid Subscribers #2

7.1 Confirmation Emails

In the previous chapter we introduced validation for the email addresses of new subscribers - they must comply with the email format.
We now have emails that are *syntactically* valid but we are still uncertain about their *existence*: does anybody actually use those email addresses? Are they reachable?
We have no idea and there is only one way to find out: sending out an actual *confirmation email*.

7.1.1 Subscriber Consent

Your spider-senses should be going off now - do we actually *need* to know at this stage of the subscriber lifetime? Can't we just wait for the next newsletter issue to find out if they receive our emails or not?

If performing thorough validation was our only concern, I'd concur: we should wait for the next issue to go out instead of adding more complexity to our `POST /subscriptions` endpoint.
There is one more thing we are concerned about though, which we cannot postpone: subscriber consent.

An email address is not a password - if you have been on the Internet long enough there is a high chance your email is not so difficult to come by.
Certain types of email addresses (e.g. professional emails) are outright public.

This opens up the possibility of *abuse*.
A malicious user could start subscribing an email address to all sort of newsletters across the internet, flooding the victim's inbox with junk.
A shady newsletter owner, instead, could start scraping email addresses from the web and adding them to its newsletter email list.

This is why a request to `POST /subscriptions` is not enough to say "This person wants to receive my newsletter content!".
For example, if you are dealing with European citizens, it is a legal requirement to get explicit consent from the user.

This is why it has become common practice to send *confirmation emails*: after entering your details

in the newsletter HTML form you will receive an email in your inbox asking you to confirm that you do indeed want to subscribe to that newsletter.
This works nicely for us - we shield our users from abuse and we get to confirm that the email addresses they provided actually exist before trying to send them a newsletter issue.

7.1.2 The Confirmation User Journey

Let's look at our confirmation flow from a user perspective.

They will receive an email with a *confirmation link*.
Once they click on it *something* happens and they are then redirected to a success page (*"You are now a subscriber of our newsletter! Yay!"*). From that point onwards, they will receive all newsletter issues in their inbox.

How will the backend work?
We will try to keep it as simple as we can - our version will not perform a redirect on confirmation, we will just return a `200 OK` to the browser.

Every time a user wants to subscribe to our newsletter they fire a `POST /subscriptions` request. Our request handler will:

- add their details to our database in the `subscriptions` table, with `status` equal to `pending_confirmation`;
- generate a (unique) `subscription_token`;
- store `subscription_token` in our database against their `id` in a `subscription_tokens` table;
- send an email to the new subscriber containing a link structured as `https://<our-api-domain>/subscriptions/confirm?token=<subscription_token>`;
- return a `200 OK`.

Once they click on the link, a browser tab will open up and a `GET` request will be fired to our `GET /subscriptions/confirm` endpoint. Our request handler will:

- retrieve `subscription_token` from the query parameters;
- retrieve the subscriber `id` associated with `subscription_token` from the `subscription_tokens` table;
- update the subscriber status from `pending_confirmation` to `active` in the `subscriptions` table;
- return a `200 OK`.

There are a few other possible designs (e.g. use a JWT instead of a unique token) and we have a few corner cases to handle (e.g. what happens if they click on the link twice? What happens if they try to subscribe twice?) - we will discuss both at the most appropriate time as we make progress with the implementation.

7.1.3 The Implementation Strategy

There is a lot to do here, so we will split the work in three conceptual chunks:

- write a module to send an email;
- adapt the logic of our existing `POST /subscriptions` request handler to match the new specification;

- write a `GET /subscriptions/confirm` request handler from scratch.

Let's get started!

7.2 `EmailClient`, Our Email Delivery Component

7.2.1 How To Send An Email

How do you *actually* send an email?
How does it work?

You have to look into SMTP - the **S**imple **M**ail **T**ransfer **P**rotocol.
It has been around since the early days of the Internet - the first RFC dates back to 1982.

SMTP does for emails what HTTP does for web pages: it is an application-level protocol that ensures that different implementations of email servers and clients can understand each other and exchange messages.

Now, let's make things clear - we will not build our own private email server, it would take too long and we would not gain much from the effort. We will be leveraging a third-party service.
What do email delivery services expect these days? Do we need to talk SMTP to them?

Not necessarily.
SMTP is a specialised protocol: unless you have been working with emails before, it is unlikely you have direct experience with it. Learning a new protocol takes time and you are bound to make mistakes along the way - that is why most providers expose two interfaces: an SMTP and a REST API.
If you are familiar with the email protocol, or you need some non-conventional configuration, go ahead with the SMTP interface. Otherwise, most developers will get up and running much faster (and more reliably) using a REST API.

As you might have guessed, that is what we will be going for as well - we will write a REST client.

7.2.1.1 Choosing An Email API

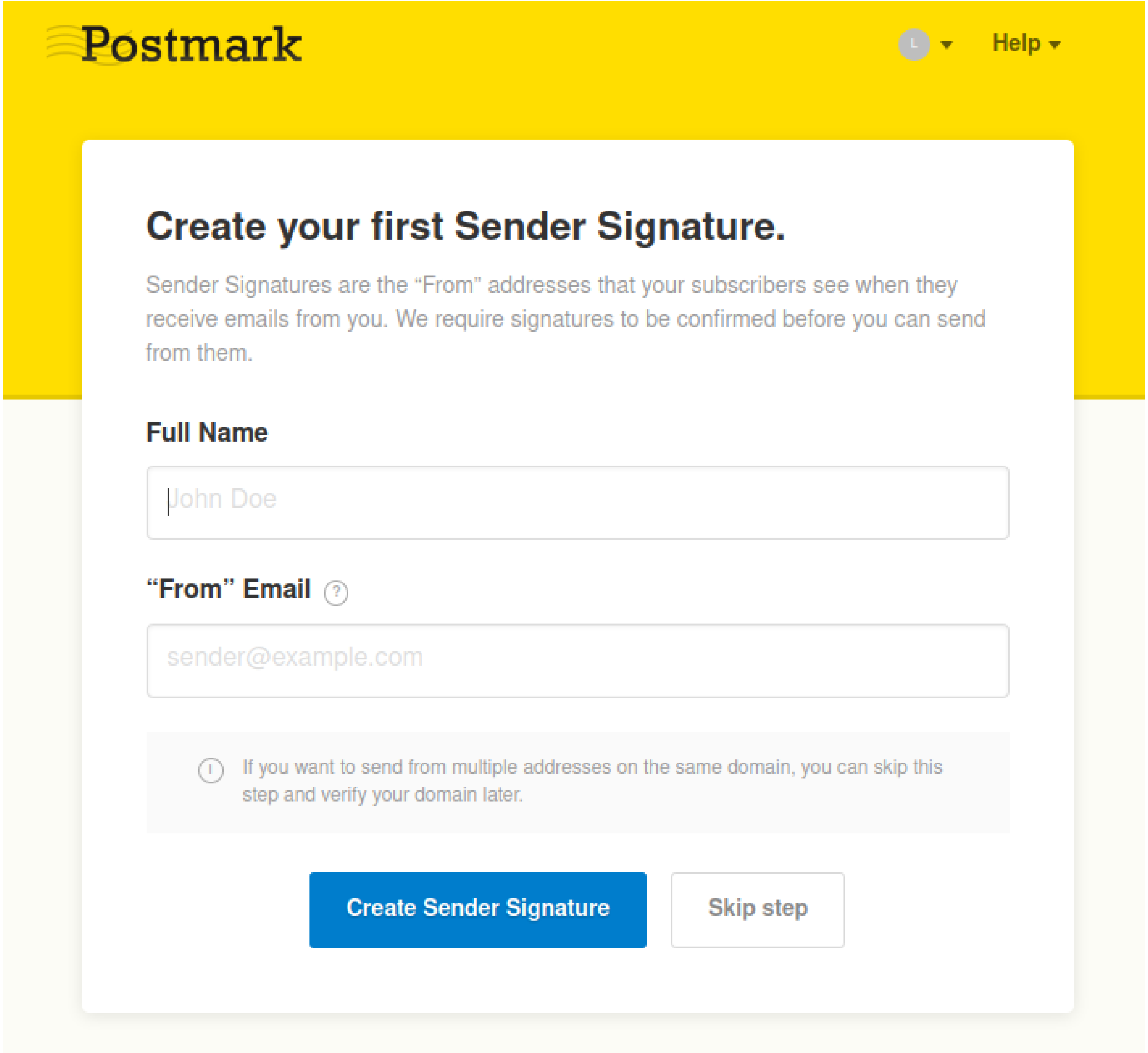
There is no shortage of email API providers on the market and you are likely to know the names of the major ones - AWS SES, SendGrid, MailGun, Mailchimp, Postmark.

I was looking for a simple enough API (e.g. how easy is it to *literally* just send an email?), a smooth onboarding flow and a free plan that does not require entering your credit card details just to test the service out.
That is how I landed on Postmark.

To complete the next sections you will have to sign up to Postmark and, once you are logged into their portal, authorise a single sender email.

Once you are done, we can move forward!

> Disclaimer: Postmark is not paying me to promote their services here.



Create your first Sender Signature.

Sender Signatures are the "From" addresses that your subscribers see when they receive emails from you. We require signatures to be confirmed before you can send from them.

Full Name

John Doe

"From" Email

sender@example.com

If you want to send from multiple addresses on the same domain, you can skip this step and verify your domain later.

Figure 7.1: Create single sender

7.2.1.2 The Email Client Interface

There are usually two approaches when it comes to a new piece of functionality: you can do it bottom-up, starting from the implementation details and slowly working your way up, or you can do it top-down, by designing the interface first and then figuring out how the implementation is going to work (to an extent).
In this case, we will go for the second route.

What kind of interface do we want for our email client?
We'd like to have some kind of `send_email` method. At the moment we just need to send a single email out at a time - we will deal with the complexity of sending emails in batches when we start working on newsletter issues.
What arguments should `send_email` accept?

We'll definitely need the recipient email address, the subject line and the email content. We'll ask for both an HTML and a plain text version of the email content - some email clients are not able to render HTML and some users explicitly disable HTML emails. By sending both versions we err on the safe side.
What about the sender email address?
We'll assume that all emails sent by an instance of the client are coming from the same address - therefore we do not need it as an argument of `send_email`, it will be one of the arguments in the constructor of the client itself.

We also expect `send_email` to be an asynchronous function, given that we will be performing I/O to talk to a remote server.

Stitching everything together, we have something that looks more or less like this:

```rust
//! src/email_client.rs

use crate::domain::SubscriberEmail;

pub struct EmailClient {
    sender: SubscriberEmail
}

impl EmailClient {
    pub async fn send_email(
        &self,
        recipient: SubscriberEmail,
        subject: &str,
        html_content: &str,
        text_content: &str
    ) -> Result<(), String> {
        todo!()
    }
}
```

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

pub mod configuration;
pub mod domain;
// New entry!
pub mod email_client;
pub mod routes;
pub mod startup;
pub mod telemetry;
```

There is an unresolved question - the return type. We sketched a `Result<(), String>` which is a way to spell *"I'll think about error handling later"*.
Plenty of work left to do, but it is a start - we said we were going to start from the interface, not that we'd nail it down in one sitting!

7.2.2 How To Write A REST Client Using `reqwest`

To talk with a REST API we need an HTTP client.
There are a few different options in the Rust ecosystem: synchronous vs asynchronous, pure Rust vs bindings to an underlying native library, tied to `tokio` or `async-std`, opinionated vs highly customisable, etc.

We will go with the most popular option on crates.io: `reqwest`.

What to say about `reqwest`?

- It has been extensively battle-tested (~8.5 million downloads);
- It offers a primarily asynchronous interface, with the option to enable a synchronous one via the `blocking` feature flag;
- It relies on `tokio` as its asynchronous executor, matching what we are already using due to `actix-web`;
- It does not depend on any system library if you choose to use `rustls` to back the TLS implementation (`rustls-tls` feature flag instead of `default-tls`), making it extremely portable.

If you look closely, we are already using `reqwest`!
It is the HTTP client we used to fire off requests at our API in the integration tests. Let's lift it from a development dependency to a runtime dependency:

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

[dependencies]
# [...]
# We need the `json` feature flag to serialize/deserialize JSON payloads
reqwest = { version = "0.11", default-features = false, features = ["json", "rustls-tls"] }

[dev-dependencies]
# Remove `reqwest`'s entry from this list
# [...]
```

7.2.2.1 `reqwest::Client`

The main type you will be dealing with when working with `reqwest` is `reqwest::Client` - it exposes all the methods we need to perform requests against a REST API.

We can get a new client instance by invoking `Client::new` or we can go with `Client::builder` if we need to tune the default configuration.
We will stick to `Client::new` for the time being.

Let's add two fields to `EmailClient`:

- `http_client`, to store a `Client` instance;
- `base_url`, to store the URL of the API we will be making requests to.

```rust
//! src/email_client.rs

use crate::domain::SubscriberEmail;
use reqwest::Client;

pub struct EmailClient {
    http_client: Client,
    base_url: String,
    sender: SubscriberEmail
}

impl EmailClient {
    pub fn new(base_url: String, sender: SubscriberEmail) -> Self {
        Self {
            http_client: Client::new(),
            base_url,
            sender
        }
    }

    // [...]
}
```

7.2.2.2 Connection Pooling

Before executing an HTTP request against an API hosted on a remote server we need to establish a connection.
It turns out that connecting is a fairly expensive operation, even more so if using HTTPS: creating a brand-new connection every time we need to fire off a request can impact the performance of our application and might lead to a problem known as *socket exhaustion* under load.

To mitigate the issue, most HTTP clients offer *connection pooling*: after the first request to a remote server has been completed, they will keep the connection open (for a certain amount of time) and re-use it if we need to fire off another request to the same server, therefore avoiding the need to re-establish a connection from scratch.

`reqwest` is no different - every time a `Client` instance is created `reqwest` initialises a connection pool under the hood.
To leverage this connection pool we need to **reuse the same `Client`** across multiple requests.
It is also worth pointing out that `Client::clone` does not create a new connection pool - we just clone a *pointer* to the underlying pool.

7.2.2.3 How To Reuse The Same `reqwest::Client` In `actix-web`

To re-use the same HTTP client across multiple requests in `actix-web` we need to store a copy of it in the application context - we will then be able to retrieve a reference to `Client` in our request handlers using an extractor (e.g. `actix_web::web::Data`).

How do we pull it off? Let's look at the code where we build a `HttpServer`:

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

pub fn run(listener: TcpListener, db_pool: PgPool) -> Result<Server, std::io::Error> {
    let db_pool = Data::new(db_pool);
    let server = HttpServer::new(move || {
        App::new()
            .wrap(TracingLogger::default())
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            .app_data(db_pool.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

We have two options:

- derive the `Clone` trait for `EmailClient`, build an instance of it once and then pass a clone to `app_data` every time we need to build an `App`:

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

#[derive(Clone)]
pub struct EmailClient {
    http_client: Client,
    base_url: String,
    sender: SubscriberEmail
}

// [...]
```

```rust
//! src/startup.rs
use crate::email_client::EmailClient;
// [...]

pub fn run(
    listener: TcpListener,
    db_pool: PgPool,
    email_client: EmailClient,
) -> Result<Server, std::io::Error> {
    let db_pool = Data::new(db_pool);
    let server = HttpServer::new(move || {
        App::new()
            .wrap(TracingLogger::default())
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            .app_data(db_pool.clone())
            .app_data(email_client.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

- wrap `EmailClient` in `actix_web::web::Data` (an `Arc` pointer) and pass a pointer to `app_data` every time we need to build an `App` - like we are doing with `PgPool`:

```
//! src/startup.rs
use crate::email_client::EmailClient;
// [...]

pub fn run(
    listener: TcpListener,
    db_pool: PgPool,
    email_client: EmailClient,
) -> Result<Server, std::io::Error> {
    let db_pool = Data::new(db_pool);
    let email_client = Data::new(email_client);
    let server = HttpServer::new(move || {
        App::new()
            .wrap(TracingLogger::default())
            .route("/health_check", web::get().to(health_check))
            .route("/subscriptions", web::post().to(subscribe))
            .app_data(db_pool.clone())
            .app_data(email_client.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

Which way is best?
If `EmailClient` were just a wrapper around a `Client` instance, the first option would be preferable - we avoid wrapping the connection pool twice with `Arc`.
This is not the case though: `EmailClient` has two data fields attached (`base_url` and `sender`). The first implementation allocates new memory to hold a copy of that data every time an `App` instance is created, while the second shares it among all `App` instances.
That's why we will be using the second strategy.

Beware though: we are creating an `App` instance for each thread - the cost of a string allocation (or a pointer clone) is negligible when looking at the bigger picture.
We are going through the decision-making process here as an exercise to understand the tradeoffs - you might have to make a similar call in the future where the cost of the two options is remarkably different.

7.2.2.4 Configuring Our `EmailClient`

If you run `cargo check`, you will get an error:

```
error[E0061]: this function takes 3 arguments but 2 arguments were supplied
  --> src/main.rs:24:5
   |
24 |     run(listener, connection_pool)?.await?;
   |     ^^^ --------  --------------- supplied 2 arguments
   |     |
   |     expected 3 arguments

error: aborting due to previous error
```

Let's fix it!
What do we have in `main` right now?

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        .connect_lazy_with(configuration.database.with_db());

    let address = format!(
        "{}:{}",
        configuration.application.host, configuration.application.port
    );
    let listener = TcpListener::bind(address)?;
    run(listener, connection_pool)?.await?;
    Ok(())
}
```

We are building the dependencies of our application using the values specified in the configuration we retrieved via `get_configuration`.
To build an `EmailClient` instance we need the base URL of the API we want to fire requests to and the sender email address - let's add them to our `Settings` struct:

```rust
//! src/configuration.rs
// [...]
use crate::domain::SubscriberEmail;

#[derive(serde::Deserialize)]
pub struct Settings {
    pub database: DatabaseSettings,
    pub application: ApplicationSettings,
    // New field!
    pub email_client: EmailClientSettings,
}

#[derive(serde::Deserialize)]
pub struct EmailClientSettings {
    pub base_url: String,
    pub sender_email: String,
}

impl EmailClientSettings {
    pub fn sender(&self) -> Result<SubscriberEmail, String> {
        SubscriberEmail::parse(self.sender_email.clone())
    }
}

// [...]
```

We then need to set values for them in our configuration files:

```yaml
#! configuration/base.yaml

application:
  # [...]
database:
# [...]
email_client:
  base_url: "localhost"
  sender_email: "test@gmail.com"
```

```yaml
#! configuration/production.yaml
application:
# [...]
database:
# [...]
email_client:
  # Value retrieved from Postmark's API documentation
  base_url: "https://api.postmarkapp.com"
  # Use the single sender email you authorised on Postmark!
  sender_email: "something@gmail.com"
```

We can now build an `EmailClient` instance in `main` and pass it to the `run` function:

```rust
//! src/main.rs
// [...]
use zero2prod::email_client::EmailClient;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        .connect_lazy_with(configuration.database.with_db());

    // Build an `EmailClient` using `configuration`
    let sender_email = configuration.email_client.sender()
        .expect("Invalid sender email address.");
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email
    );

    let address = format!(
        "{}:{}",
        configuration.application.host, configuration.application.port
    );
    let listener = TcpListener::bind(address)?;
    // New argument for `run`, `email_client`
    run(listener, connection_pool, email_client)?.await?;
    Ok(())
}
```

`cargo check` should now pass, although there are a few warnings about unused variables - we will get to those soon enough.
What about our tests?

`cargo check --all-targets` returns a similar error to the one we were seeing before with `cargo check`:

```
error[E0061]: this function takes 3 arguments but 2 arguments were supplied
  --> tests/health_check.rs:36:18
   |
36 |     let server = run(listener, connection_pool.clone())
   |                  ^^^ --------  ----------------------- supplied 2 arguments
   |                  |
   |                  expected 3 arguments

error: aborting due to previous error
```

You are right - it is a symptom of code duplication. We will get to refactor the initialisation logic of our integration tests, but not yet.
Let's patch it quickly to make it compile:

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

// [...]
use zero2prod::email_client::EmailClient;
// [...]

async fn spawn_app() -> TestApp {
    // [...]

    let mut configuration = get_configuration()
        .expect("Failed to read configuration.");
    configuration.database.database_name = Uuid::new_v4().to_string();
    let connection_pool = configure_database(&configuration.database).await;

    // Build a new email client
    let sender_email = configuration.email_client.sender()
        .expect("Invalid sender email address.");
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email
    );

    // Pass the new client to `run`!
    let server = run(listener, connection_pool.clone(), email_client)
        .expect("Failed to bind address");
    let _ = tokio::spawn(server);
    TestApp {
        address,
        db_pool: connection_pool,
    }
}

// [...]
```

`cargo test` should succeed now.

7.2.3 How To Test A REST Client

We have gone through most of the setup steps: we sketched an interface for `EmailClient` and we wired it up with the application, using a new configuration type - `EmailClientSettings`.
To stay true to our test-driven development approach, it is now time to write a test!
We could start from our integration tests: change the ones for `POST /subscriptions` to make sure that the endpoint conforms to our new requirements.
It would take us a long time to turn them green though: apart from sending an email, we need to add logic to generate a unique token and store it.

Let's start smaller: we will just test our `EmailClient` component in isolation.
It will boost our confidence that it behaves as expected when tested as a *unit*, reducing the number of issues we might encounter when integrating it into the larger confirmation email flow.
It will also give us a chance to see if the interface we landed on is ergonomic and easy to test.

What should we actually test though?
The main purpose of our `EmailClient::send_email` is to perform an HTTP call: how do we know if it happened? How do we check that the body and the headers were populated as we expected?
We need to *intercept* that HTTP request - time to spin up a mock server!

7.2.3.1 HTTP Mocking With `wiremock`

Let's add a new module for tests at the bottom of `src/email_client.rs` with the skeleton of a new test:

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

#[cfg(test)]
mod tests {
    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        todo!()
    }
}
```

This will not compile straight-away - we need to add two feature flags to `tokio` in our `Cargo.toml`:

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

[dev-dependencies]
# [...]
tokio = { version = "1", features = ["rt", "macros"] }
```

We do not know enough about Postmark to make assertions about what we should see in the outgoing HTTP request.
Nonetheless, as the test name says, it is reasonable to expect a request to be fired to the server at `EmailClient::base_url`!

Let's add `wiremock` to our development dependencies:

```toml
#! Cargo.toml
```

```toml
# [...]

[dev-dependencies]
# [...]
wiremock = "0.5"
```

Using `wiremock`, we can write `send_email_fires_a_request_to_base_url` as follows:

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

#[cfg(test)]
mod tests {
    use crate::domain::SubscriberEmail;
    use crate::email_client::EmailClient;
    use fake::faker::internet::en::SafeEmail;
    use fake::faker::lorem::en::{Paragraph, Sentence};
    use fake::{Fake, Faker};
    use wiremock::matchers::any;
    use wiremock::{Mock, MockServer, ResponseTemplate};

    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        // Arrange
        let mock_server = MockServer::start().await;
        let sender = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let email_client = EmailClient::new(mock_server.uri(), sender);

        Mock::given(any())
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        let subscriber_email = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let subject: String = Sentence(1..2).fake();
        let content: String = Paragraph(1..10).fake();

        // Act
        let _ = email_client
            .send_email(subscriber_email, &subject, &content, &content)
            .await;

        // Assert
    }
}
```

Let's break down what is happening, step by step.

7.2.3.2 `wiremock::MockServer`

```rust
let mock_server = MockServer::start().await;
```

`wiremock::MockServer` is a full-blown HTTP server.
`MockServer::start` asks the operating system for a random available port and spins up the server on a background thread, ready to listen for incoming requests.

How do we point our email client to our mock server? We can retrieve the address of the mock server using the `MockServer::uri` method; we can then pass it as `base_url` to `EmailClient::new`:

```
let email_client = EmailClient::new(mock_server.uri(), sender);
```

7.2.3.3 `wiremock::Mock`

Out of the box, `wiremock::MockServer` returns `404 Not Found` to all incoming requests.
We can instruct the mock server to behave differently by mounting a `Mock`.

```
Mock::given(any())
    .respond_with(ResponseTemplate::new(200))
    .expect(1)
    .mount(&mock_server)
    .await;
```

When `wiremock::MockServer` receives a request, it iterates over all the mounted mocks to check if the request *matches* their conditions.
The matching conditions for a mock are specified using `Mock::given`.

We are passing `any()` to `Mock::Given` which, according to `wiremock`'s documentation,

> Match all incoming requests, regardless of their method, path, headers or body. You can use it to verify that a request has been fired towards the server, without making any other assertion about it.

Basically, it always matches, regardless of the request - which is what we want here!

When an incoming request matches the conditions of a mounted mock, `wiremock::MockServer` returns a response following what was specified in `respond_with`.
We passed `ResponseTemplate::new(200)` - a `200 OK` response without a body.

A `wiremock::Mock` becomes effective only after it has been mounted on a `wiremock::Mockserver` - that's what our call to `Mock::mount` is about.

7.2.3.4 The Intent Of A Test Should Be Clear

We then have the actual invocation of `EmailClient::send_email`:

```
let subscriber_email = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
let subject: String = Sentence(1..2).fake();
let content: String = Paragraph(1..10).fake();

// Act
let _ = email_client
    .send_email(subscriber_email, &subject, &content, &content)
    .await;
```

You'll notice that we are leaning heavily on `fake` here: we are generating random data for all the inputs to `send_email` (and `sender`, in the previous section).
We could have just hard-coded a bunch of values, why did we choose to go all the way and make them random?

A reader, skimming the test code, should be able to identify easily the property that we are trying to test.

Using random data conveys a specific message: do not pay attention to these inputs, their values do not influence the outcome of the test, that's why they are random!

Hard-coded values, instead, should always give you pause: does it matter that `subscriber_email` is set to `marco@gmail.com`? Should the test pass if I set it to another value?
In a test like ours, the answer is obvious. In a more intricate setup, it often isn't.

7.2.3.5 Mock expectations

The end of the test looks a bit cryptic: there is an `// Assert` comment... but no assertion afterwards.
Let's go back to our `Mock` setup line:

```
Mock::given(any())
    .respond_with(ResponseTemplate::new(200))
    .expect(1)
    .mount(&mock_server)
    .await;
```

What does `.expect(1)` do?
It sets an *expectation* on our mock: we are telling the mock server that during this test it should receive *exactly* one request that matches the conditions set by this mock.
We could also use ranges for our expectations - e.g. `expect(1..)` if we want to see *at least* one request, `expect(1..=3)` if we expect at least one request but no more than three, etc.

Expectations are verified when `MockServer` goes out of scope - at the end of our test function, indeed!
Before shutting down, `MockServer` will iterate over all the mounted mocks and check if their expectations have been verified. If the verification step fails, it will trigger a panic (and fail the test).

Let's run `cargo test`:

```
---- email_client::tests::send_email_fires_a_request_to_base_url stdout ----
thread 'email_client::tests::send_email_fires_a_request_to_base_url' panicked at
'not yet implemented', src/email_client.rs:24:9
```

Ok, we are not even getting to the end of the test yet because we have a placeholder `todo!()` as the body of `send_email`.
Let's replace it with a dummy `Ok`:

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

impl EmailClient {
    // [...]

    pub async fn send_email(
        &self,
        recipient: SubscriberEmail,
        subject: &str,
        html_content: &str,
        text_content: &str
    ) -> Result<(), String> {
        // No matter the input
        Ok(())
    }
}

// [...]
```

If we run `cargo test` again, we'll get to see `wiremock` in action:

```
---- email_client::tests::send_email_fires_a_request_to_base_url stdout ----
thread 'email_client::tests::send_email_fires_a_request_to_base_url' panicked at
'Verifications failed:
- Mock #0.
        Expected range of matching incoming requests: == 1
        Number of matched incoming requests: 0
'
```

The server expected one request, but it received none - therefore the test failed.

The time has come to properly flesh out `EmailClient::send_email`.

7.2.4 First Sketch Of `EmailClient::send_email`

To implement `EmailClient::send_email` we need to check out the API documentation of Postmark. Let's start from their "Send a single email" user guide.

Their email sending example looks like this:

```
curl "https://api.postmarkapp.com/email" \
  -X POST \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -H "X-Postmark-Server-Token: server token" \
  -d '{
  "From": "sender@example.com",
  "To": "receiver@example.com",
  "Subject": "Postmark test",
  "TextBody": "Hello dear Postmark user.",
  "HtmlBody": "<html><body><strong>Hello</strong> dear Postmark user.</body></html>"
}'
```

Let's break it down - to send an email we need:

- a `POST` request to the `/email` endpoint;
- a JSON body, with fields that map closely to the arguments of `send_email`. We need to be careful with field names, they must be pascal cased;
- an authorization header, `X-Postmark-Server-Token`, with a value set to a secret token that we can retrieve from their portal.

If the request succeeds, we get something like this back:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
    "To": "receiver@example.com",
    "SubmittedAt": "2021-01-12T07:25:01.4178645-05:00",
    "MessageID": "0a129aee-e1cd-480d-b08d-4f48548ff48d",
    "ErrorCode": 0,
    "Message": "OK"
}
```

We have enough to implement the happy path!

7.2.4.1 `reqwest::Client::post`

`reqwest::Client` exposes a `post` method - it takes the URL we want to call with a `POST` request as argument and it returns a `RequestBuilder`.
`RequestBuilder` gives us a fluent API to build out the rest of the request we want to send, piece by piece.
Let's give it a go:

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

impl EmailClient {
    // [...]

    pub async fn send_email(
        &self,
        recipient: SubscriberEmail,
        subject: &str,
        html_content: &str,
        text_content: &str
    ) -> Result<(), String> {
        // You can do better using `reqwest::Url::join` if you change
        // `base_url`'s type from `String` to `reqwest::Url`.
        // I'll leave it as an exercise for the reader!
        let url = format!("{}/email", self.base_url);
        let builder = self.http_client.post(&url);
        Ok(())
    }
}

// [...]
```

7.2.4.2 JSON body

We can encode the request body schema as a struct:

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

impl EmailClient {
    // [...]

    pub async fn send_email(
        &self,
        recipient: SubscriberEmail,
        subject: &str,
        html_content: &str,
        text_content: &str
    ) -> Result<(), String> {
        let url = format!("{}/email", self.base_url);
        let request_body = SendEmailRequest {
            from: self.sender.as_ref().to_owned(),
            to: recipient.as_ref().to_owned(),
            subject: subject.to_owned(),
            html_body: html_content.to_owned(),
            text_body: text_content.to_owned(),
        };
        let builder = self.http_client.post(&url);
        Ok(())
    }
}

struct SendEmailRequest {
    from: String,
    to: String,
    subject: String,
    html_body: String,
    text_body: String,
}

// [...]
```

If the `json` feature flag for `reqwest` is enabled (as we did), `builder` will expose a `json` method that we can leverage to set `request_body` as the JSON body of the request:

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

impl EmailClient {
// [...]

    pub async fn send_email(
        &self,
        recipient: SubscriberEmail,
        subject: &str,
        html_content: &str,
        text_content: &str
    ) -> Result<(), String> {
        let url = format!("{}/email", self.base_url);
        let request_body = SendEmailRequest {
            from: self.sender.as_ref().to_owned(),
            to: recipient.as_ref().to_owned(),
            subject: subject.to_owned(),
            html_body: html_content.to_owned(),
            text_body: text_content.to_owned(),
        };
        let builder = self.http_client.post(&url).json(&request_body);
        Ok(())
    }
}
```

It *almost* works:

```
error[E0277]: the trait bound `SendEmailRequest: Serialize` is not satisfied
  --> src/email_client.rs:34:56
   |
34 |         let builder = self.http_client.post(&url).json(&request_body);
   |                                                        ^^^^^^^^^^^^^
```

```
          the trait `Serialize` is not implemented for `SendEmailRequest`
```

Let's derive `serde::Serialize` for `SendEmailRequest` to make it serializable:

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

#[derive(serde::Serialize)]
struct SendEmailRequest {
    from: String,
    to: String,
    subject: String,
    html_body: String,
    text_body: String,
}
```

Awesome, it compiles!
The `json` method goes a bit further than simple serialization: it will also set the `Content-Type` header to `application/json` - matching what we saw in the example!

7.2.4.3 Authorization Token

We are almost there - we need to add an authorization header, `X-Postmark-Server-Token`, to the request.
Just like the sender email address, we want to store the token value as a field in `EmailClient`.

Let's amend `EmailClient::new` and `EmailClientSettings`:

```rust
//! src/email_client.rs
use secrecy::Secret;
// [...]

pub struct EmailClient {
    // [...]
    // We don't want to log this by accident
    authorization_token: Secret<String>
}

impl EmailClient {
    pub fn new(
        // [...]
        authorization_token: Secret<String>
    ) -> Self {
        Self {
            // [...]
            authorization_token
        }
    }

    // [...]
}
```

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

#[derive(serde::Deserialize)]
```

```rust
pub struct EmailClientSettings {
    // [...]
    // New (secret) configuration value!
    pub authorization_token: Secret<String>
}

// [...]
```

We can then let the compiler tell us what else needs to be modified:

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

#[cfg(test)]
mod tests {
    use secrecy::Secret;
    // [...]

    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        let mock_server = MockServer::start().await;
        let sender = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        // New argument!
        let email_client = EmailClient::new(
            mock_server.uri(),
            sender,
            Secret::new(Faker.fake())
        );
        // [...]
    }
}
```

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        // Pass argument from configuration
        configuration.email_client.authorization_token,
    );
    // [...]
}
```

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

async fn spawn_app() -> TestApp {
    // [...]
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        // Pass argument from configuration
        configuration.email_client.authorization_token,
    );
    // [...]
}
// [...]
```

```yaml
#! configuration/base.yml
# [...]
email_client:
  base_url: "localhost"
  sender_email: "test@gmail.com"
  # New value!
  # We are only setting the development value,
  # we'll deal with the production token outside of version control
  # (given that it's a sensitive secret!)
  authorization_token: "my-secret-token"
```

We can now use the authorization token in `send_email`:

```rust
//! src/email_client.rs
use secrecy::{ExposeSecret, Secret};
// [...]

impl EmailClient {
// [...]

    pub async fn send_email(
        // [...]
    ) -> Result<(), String> {
        // [...]
        let builder = self
            .http_client
            .post(&url)
            .header(
                "X-Postmark-Server-Token",
                self.authorization_token.expose_secret()
            )
            .json(&request_body);
        Ok(())
    }
}
```

It compiles straight away.

7.2.4.4 Executing The Request

We have all the ingredients - we just need to fire the request now!
We can use the `send` method:

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

impl EmailClient {
// [...]

    pub async fn send_email(
        // [...]
    ) -> Result<(), String> {
        // [...]
        self
            .http_client
            .post(&url)
            .header(
                "X-Postmark-Server-Token",
                self.authorization_token.expose_secret()
            )
            .json(&request_body)
            .send()
            .await?;
        Ok(())
    }
}
```

`send` is asynchronous, therefore we need to `await` the future it returns.
`send` is also a fallible operation - e.g. we might fail to establish a connection to the server. We'd like to return an error if `send` fails - that's why we use the `?` operator.

The compiler, though, is not happy:

```
error[E0277]: `?` couldn't convert the error to `std::string::String`
--> src/email_client.rs:41:19
   |
41 |             .await?;
   |                   ^
    the trait `From<reqwest::Error>` is not implemented for `std::string::String`
```

The error variant returned by `send` is of type `reqwest::Error`, while our `send_email` uses `String` as error type. The compiler has looked for a conversion (an implementation of the `From` trait), but it could not find any - therefore it errors out.
If you recall, we used `String` as error variant mostly as a placeholder - let's change `send_email`'s signature to return `Result<(), reqwest::Error>`.

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

impl EmailClient {
// [...]

    pub async fn send_email(
        // [...]
    ) -> Result<(), reqwest::Error> {
        // [...]
    }
}
```

The error should be gone now!
`cargo test` should pass too: congrats!

7.2.5 Tightening Our Happy Path Test

Let's look again at our "happy path" test:

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

#[cfg(test)]
mod tests {
    use crate::domain::SubscriberEmail;
    use crate::email_client::EmailClient;
    use fake::faker::internet::en::SafeEmail;
    use fake::faker::lorem::en::{Paragraph, Sentence};
    use fake::{Fake, Faker};
    use wiremock::matchers::any;
    use wiremock::{Mock, MockServer, ResponseTemplate};

    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        // Arrange
        let mock_server = MockServer::start().await;
        let sender = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let email_client = EmailClient::new(
            mock_server.uri(),
            sender,
            Secret::new(Faker.fake())
        );

        let subscriber_email = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let subject: String = Sentence(1..2).fake();
        let content: String = Paragraph(1..10).fake();

        Mock::given(any())
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        // Act
        let _ = email_client
            .send_email(subscriber_email, &subject, &content, &content)
            .await;

        // Assert
        // Mock expectations are checked on drop
    }
}
```

To ease ourselves into the world of `wiremock` we started with something very basic - we are just asserting that the mock server gets called once. Let's beef it up to check that the outgoing request looks indeed like we expect it to.

7.2.5.0.1 Headers, Path And Method

`any` is not the only matcher offered by `wiremock` out of the box: there are handful available in `wiremock`'s `matchers` module.

We can use `header_exists` to verify that the `X-Postmark-Server-Token` is set on the request to the server:

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

```rust
#[cfg(test)]
mod tests {
    // [...]
    // We removed `any` from the import list
    use wiremock::matchers::header_exists;

    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        // [...]

        Mock::given(header_exists("X-Postmark-Server-Token"))
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        // [...]
    }
}
```

We can chain multiple matchers together using the `and` method.
Let's add `header` to check that the `Content-Type` is set to the correct value, `path` to assert on the endpoint being called and `method` to verify the HTTP verb:

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

#[cfg(test)]
mod tests {
    // [...]
    use wiremock::matchers::{header, header_exists, path, method};

    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        // [...]

        Mock::given(header_exists("X-Postmark-Server-Token"))
            .and(header("Content-Type", "application/json"))
            .and(path("/email"))
            .and(method("POST"))
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        // [...]
    }
}
```

7.2.5.0.2 Body

So far, so good: `cargo test` still passes.
What about the request body?

We could use `body_json` to match *exactly* the request body.
We probably do not need to go as far as that - it would be enough to check that the body is valid JSON and it contains the set of field names shown in Postmark's example.

There is no out-of-the-box matcher that suits our needs - we need to implement our own!
`wiremock` exposes a `Match` trait - everything that implements it can be used as a matcher in `given` and `and`.
Let's stub it out:

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

#[cfg(test)]
mod tests {
    use wiremock::Request;
    // [...]

    struct SendEmailBodyMatcher;

    impl wiremock::Match for SendEmailBodyMatcher {
        fn matches(&self, request: &Request) -> bool {
            unimplemented!()
        }
    }

    // [...]
}
```

We get the incoming request as input, `request`, and we need to return a boolean value as output: `true`, if the mock matched, `false` otherwise.
We need to deserialize the request body as JSON - let's add `serde-json` to the list of our development dependencies:

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

[dev-dependencies]
# [...]
serde_json = "1"
```

We can now write `matches`' implementation:

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

#[cfg(test)]
mod tests {
    // [...]

    struct SendEmailBodyMatcher;

    impl wiremock::Match for SendEmailBodyMatcher {
        fn matches(&self, request: &Request) -> bool {
            // Try to parse the body as a JSON value
            let result: Result<serde_json::Value, _> =
                serde_json::from_slice(&request.body);
            if let Ok(body) = result {
                // Check that all the mandatory fields are populated
                // without inspecting the field values
                body.get("From").is_some()
```

```
                    && body.get("To").is_some()
                    && body.get("Subject").is_some()
                    && body.get("HtmlBody").is_some()
                    && body.get("TextBody").is_some()
            } else {
                // If parsing failed, do not match the request
                false
            }
        }
    }

    #[tokio::test]
    async fn send_email_fires_a_request_to_base_url() {
        // [...]

        Mock::given(header_exists("X-Postmark-Server-Token"))
            .and(header("Content-Type", "application/json"))
            .and(path("/email"))
            .and(method("POST"))
            // Use our custom matcher!
            .and(SendEmailBodyMatcher)
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        // [...]
    }
}
```

It compiles!
But our tests are failing now…

```
---- email_client::tests::send_email_fires_a_request_to_base_url stdout ----
thread 'email_client::tests::send_email_fires_a_request_to_base_url' panicked at
'Verifications failed:
- Mock #0.
        Expected range of matching incoming requests: == 1
        Number of matched incoming requests: 0
'
```

Why is that?
Let's add a `dbg!` statement to our matcher to inspect the incoming request:

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

#[cfg(test)]
mod tests {
    // [...]

    impl wiremock::Match for SendEmailBodyMatcher {
        fn matches(&self, request: &Request) -> bool {
            // [...]
            if let Ok(body) = result {
                dbg!(&body);
                // [...]
            } else {
                false
            }
        }
    }
    // [...]
}
```

If you run the test again with `cargo test send_email` you will get something that looks like this:

```
--- email_client::tests::send_email_fires_a_request_to_base_url stdout ----
[src/email_client.rs:71] &body = Object({
    "from": String("[...]"),
    "to": String("[...]"),
    "subject": String("[...]"),
    "html_body": String("[...]"),
    "text_body": String("[...]"),
})
thread 'email_client::tests::send_email_fires_a_request_to_base_url' panicked at '
Verifications failed:
- Mock #0.
        Expected range of matching incoming requests: == 1
        Number of matched incoming requests: 0
'
```

It seems we forgot about the casing requirement - field names must be pascal cased!
We can fix it easily by adding an annotation on `SendEmailRequest`:

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

#[derive(serde::Serialize)]
#[serde(rename_all = "PascalCase")]
struct SendEmailRequest {
    from: String,
    to: String,
    subject: String,
    html_body: String,
    text_body: String,
}
```

The test should pass now.
Before we move on, let's rename the test to `send_email_sends_the_expected_request` - it captures better the test purpose at this point.

7.2.5.1 Refactoring: Avoid Unnecessary Memory Allocations

We focused on getting `send_email` to work - now we can look at it again to see if there is any room for improvement.
Let's zoom on the request body:

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

impl EmailClient {
// [...]
```

```rust
    pub async fn send_email(
        // [...]
    ) -> Result<(), reqwest::Error> {
        // [...]
        let request_body = SendEmailRequest {
            from: self.sender.as_ref().to_owned(),
            to: recipient.as_ref().to_owned(),
            subject: subject.to_owned(),
            html_body: html_content.to_owned(),
            text_body: text_content.to_owned(),
        };
        // [...]
    }
}

#[derive(serde::Serialize)]
#[serde(rename_all = "PascalCase")]
struct SendEmailRequest {
    from: String,
    to: String,
    subject: String,
    html_body: String,
    text_body: String,
}
```

For each field we are allocating a bunch of new memory to store a cloned `String` - it is wasteful. It would be more efficient to reference the existing data without performing any additional allocation.
We can pull it off by restructuring `SendEmailRequest`: instead of `String` we have to use a string slice (`&str`) as type for all fields.
A string slice is a just pointer to a memory buffer owned by somebody else. To store a reference in a struct we need to add a lifetime parameter: it keeps track of how long those references are valid for - it's the compiler's job to make sure that references do not stay around longer than the memory buffer they point to!

Let's do it!

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

impl EmailClient {
// [...]

    pub async fn send_email(
        // [...]
    ) -> Result<(), reqwest::Error> {
        // [...]
        // No more `.to_owned`!
        let request_body = SendEmailRequest {
            from: self.sender.as_ref(),
            to: recipient.as_ref(),
            subject,
            html_body: html_content,
            text_body: text_content,
        };
        // [...]
    }
}

#[derive(serde::Serialize)]
#[serde(rename_all = "PascalCase")]
// Lifetime parameters always start with an apostrophe, `'`
struct SendEmailRequest<'a> {
    from: &'a str,
    to: &'a str,
    subject: &'a str,
    html_body: &'a str,
    text_body: &'a str,
}
```

That's it, quick and painless - `serde` does all the heavy lifting for us and we are left with more performant code!

7.2.6 Dealing With Failures

We have a good grip on the happy path - what happens instead if things don't go as expected?
We will look at two scenarios:

- non-success status codes (e.g. `4xx`, `5xx`, etc.);
- slow responses.

7.2.6.1 Error Status Codes

Our current happy path test is only making assertions on the side-effect performed by `send_email` - we are not actually inspecting the value it returns!
Let's make sure that it is an `Ok(())` if the server returns a `200 OK`:

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

#[cfg(test)]
mod tests {
    // [...]
    use wiremock::matchers::any;
    use claim::assert_ok;
    // [...]

    // New happy-path test!
    #[tokio::test]
    async fn send_email_succeeds_if_the_server_returns_200() {
        // Arrange
        let mock_server = MockServer::start().await;
        let sender = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let email_client = EmailClient::new(
            mock_server.uri(),
            sender,
            Secret::new(Faker.fake())
        );

        let subscriber_email = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let subject: String = Sentence(1..2).fake();
        let content: String = Paragraph(1..10).fake();

        // We do not copy in all the matchers we have in the other test.
        // The purpose of this test is not to assert on the request we
        // are sending out!
        // We add the bare minimum needed to trigger the path we want
        // to test in `send_email`.
        Mock::given(any())
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        // Act
        let outcome = email_client
            .send_email(subscriber_email, &subject, &content, &content)
            .await;

        // Assert
        assert_ok!(outcome);
    }
}
```

No surprises, the test passes.
Let's look at the opposite case now - we expect an `Err` variant if the server returns a `500 Internal Server Error`.

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

#[cfg(test)]
mod tests {
    // [...]
    use claim::assert_err;
    // [...]

    #[tokio::test]
    async fn send_email_fails_if_the_server_returns_500() {
        // Arrange
        let mock_server = MockServer::start().await;
        let sender = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let email_client = EmailClient::new(
            mock_server.uri(),
            sender,
            Secret::new(Faker.fake())
        );

        let subscriber_email = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let subject: String = Sentence(1..2).fake();
        let content: String = Paragraph(1..10).fake();

        Mock::given(any())
            // Not a 200 anymore!
            .respond_with(ResponseTemplate::new(500))
            .expect(1)
            .mount(&mock_server)
            .await;

        // Act
        let outcome = email_client
            .send_email(subscriber_email, &subject, &content, &content)
            .await;

        // Assert
        assert_err!(outcome);
    }
}
```

We got some work to do here instead:

```
--- email_client::tests::send_email_fails_if_the_server_returns_500 stdout ----
thread 'email_client::tests::send_email_fails_if_the_server_returns_500' panicked at
'assertion failed, expected Err(..), got Ok(())', src/email_client.rs:163:9
```

Let's look again at `send_email`:

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

impl EmailClient {
    //[...]
    pub async fn send_email(
        //[...]
    ) -> Result<(), reqwest::Error> {
        //[...]
        self.http_client
            .post(&url)
            .header(
                "X-Postmark-Server-Token",
                self.authorization_token.expose_secret()
            )
            .json(&request_body)
            .send()
            .await?;
        Ok(())
    }
}
//[...]
```

The only step that might return an error is `send` - let's check `reqwest`'s docs!

> This method fails if there was an error while sending request, redirect loop was detected or redirect limit was exhausted.

Basically, `send` returns `Ok` as long as it gets a valid response from the server - no matter the status code!
To get the behaviour we want we need to look at the methods available on `reqwest::Response` - in particular, `error_for_status`:

> Turn a response into an error if the server returned an error.

It seems to suit our needs, let's try it out.

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

impl EmailClient {
    //[...]
    pub async fn send_email(
        //[...]
    ) -> Result<(), reqwest::Error> {
        //[...]
        self.http_client
            .post(&url)
            .header(
                "X-Postmark-Server-Token",
                self.authorization_token.expose_secret()
            )
            .json(&request_body)
            .send()
            .await?
            .error_for_status()?;
        Ok(())
    }
}
//[...]
```

Awesome, the test passes!

7.2.6.2 Timeouts

What happens instead if the server returns a `200 OK`, but it takes *ages* to send it back?
We can instruct our mock server to wait a configurable amount of time before sending a response back.
Let's experiment a little with a new integration test - what if the server takes **3 minutes** to respond!?

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

#[cfg(test)]
mod tests {
    // [...]

    #[tokio::test]
    async fn send_email_times_out_if_the_server_takes_too_long() {
        // Arrange
        let mock_server = MockServer::start().await;
        let sender = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let email_client = EmailClient::new(
            mock_server.uri(),
            sender,
            Secret::new(Faker.fake())
        );

        let subscriber_email = SubscriberEmail::parse(SafeEmail().fake()).unwrap();
        let subject: String = Sentence(1..2).fake();
        let content: String = Paragraph(1..10).fake();

        let response = ResponseTemplate::new(200)
            // 3 minutes!
            .set_delay(std::time::Duration::from_secs(180));
        Mock::given(any())
            .respond_with(response)
            .expect(1)
            .mount(&mock_server)
            .await;

        // Act
        let outcome = email_client
            .send_email(subscriber_email, &subject, &content, &content)
            .await;

        // Assert
        assert_err!(outcome);
    }
}
```

After a while, you should see something like this:

```
test email_client::tests::send_email_times_out_if_the_server_takes_too_long ...
test email_client::tests::send_email_times_out_if_the_server_takes_too_long
has been running for over 60 seconds
```

This is far from ideal: if the server starts misbehaving we might start to accumulate several "hanging" requests.
We are not hanging up on the server, so the connection is busy: every time we need to send an email we will have to open a new connection. If the server does not recover fast enough, and we do not close any of the open connections, we might end up with socket exhaustion/performance degradation.

As a rule of thumb: every time you are performing an IO operation, *always* set a timeout!
If the server takes longer than the timeout to respond, we should fail and return an error.

Choosing the right timeout value is often more an art than a science, especially if retries are involved: set it too low and you might overwhelm the server with retried requests; set it too high and you risk again to see degradation on the client side.
Nonetheless, better to have a conservative timeout threshold than to have none.

`reqwest` gives us two options: we can either add a default timeout on the `Client` itself, which applies to all outgoing requests, or we can specify a per-request timeout.
Let's go for a `Client`-wide timeout: we'll set it in `EmailClient::new`.

```rust
//! src/email_client.rs
// [...]
impl EmailClient {
    pub fn new(
        // [...]
    ) -> Self {
        let http_client = Client::builder()
            .timeout(std::time::Duration::from_secs(10))
            .build()
            .unwrap();
        Self {
            http_client,
            base_url,
            sender,
            authorization_token,
        }
    }
}
// [...]
```

If we run the test again, it should pass (after 10 seconds have elapsed).

7.2.6.3 Refactoring: Test Helpers

There is a lot of duplicated code in our four tests for `EmailClient` - let's extract the common bits in a set of test helpers.

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

#[cfg(test)]
mod tests {
    // [...]

    /// Generate a random email subject
    fn subject() -> String {
        Sentence(1..2).fake()
    }

    /// Generate a random email content
    fn content() -> String {
        Paragraph(1..10).fake()
    }

    /// Generate a random subscriber email
    fn email() -> SubscriberEmail {
        SubscriberEmail::parse(SafeEmail().fake()).unwrap()
    }

    /// Get a test instance of `EmailClient`.
    fn email_client(base_url: String) -> EmailClient {
        EmailClient::new(base_url, email(), Secret::new(Faker.fake()))
    }

    // [...]
}
```

Let's use them in `send_email_sends_the_expected_request`:

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

#[cfg(test)]
mod tests {
    // [...]
```

```rust
    #[tokio::test]
    async fn send_email_sends_the_expected_request() {
        // Arrange
        let mock_server = MockServer::start().await;
        let email_client = email_client(mock_server.uri());

        Mock::given(header_exists("X-Postmark-Server-Token"))
            .and(header("Content-Type", "application/json"))
            .and(path("/email"))
            .and(method("POST"))
            .and(SendEmailBodyMatcher)
            .respond_with(ResponseTemplate::new(200))
            .expect(1)
            .mount(&mock_server)
            .await;

        // Act
        let _ = email_client
            .send_email(email(), &subject(), &content(), &content())
            .await;

        // Assert
    }
}
```

Way less visual noise - the intent of the test is front and center.
Go ahead and refactor the other three!

7.2.6.4 Refactoring: Fail fast

The timeout on our HTTP client is currently hard-coded to 10 seconds:

```rust
//! src/email_client.rs
// [...]
impl EmailClient {
    pub fn new(
        // [...]
    ) -> Self {
        let http_client = Client::builder()
            .timeout(std::time::Duration::from_secs(10))
        // [...]
    }
}
```

This implies that our timeout test takes roughly 10 seconds to fail - that is a long time, especially if you are running tests after every little change.

Let's make the timeout threshold configurable to keep our test suite responsive.

```rust
//! src/email_client.rs
// [...]
impl EmailClient {
    pub fn new(
        // [...]
        // New argument!
        timeout: std::time::Duration,
    ) -> Self {
        let http_client = Client::builder()
            .timeout(timeout)
        // [...]
    }
}
```

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

#[derive(serde::Deserialize)]
pub struct EmailClientSettings {
    // [...]
    // New configuration value!
    pub timeout_milliseconds: u64
}

impl EmailClientSettings {
    // [...]
    pub fn timeout(&self) -> std::time::Duration {
        std::time::Duration::from_millis(self.timeout_milliseconds)
    }
}
```

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

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let timeout = configuration.email_client.timeout();
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
        // Pass new argument from configuration
        timeout
    );
    // [...]
}
```

```yaml
#! configuration/base.yaml
# [...]
email_client:
  # [...]
  timeout_milliseconds: 10000
```

The project should compile.
We still need to edit the tests though!

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

#[cfg(test)]
mod tests {
    // [...]
    fn email_client(base_url: String) -> EmailClient {
        EmailClient::new(
            base_url,
            email(),
            Secret::new(Faker.fake()),
            // Much lower than 10s!
            std::time::Duration::from_millis(200),
        )
    }
}
```

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

async fn spawn_app() -> TestApp {
    // [...]
    let timeout = configuration.email_client.timeout();
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
        timeout
    );
}
```

All tests should succeed - and the overall execution time should be down to less than a second for the whole test suite.

7.3 Skeleton And Principles For A Maintainable Test Suite

It took us a bit of work, but we now have a pretty decent REST client for Postmark's API.

`EmailClient` is just the first ingredient for our confirmation email flow: we have yet to find a way to generate unique confirmation links, which we will then have to embed in the body of the outgoing confirmation emails.
Both tasks will have to wait a bit longer.

We have used a test-driven approach to write all new pieces of functionality throughout the book. While this strategy has served us well, we have not invested a lot of time into *refactoring* our test code. As a result, our `tests` folder is a bit of mess at this point.
Before moving forward, we will restructure our integration test suite to support us as our application grows in complexity and the number of tests increases.

7.3.1 Why Do We Write Tests?

Is writing tests a good use of developers' time?

A good test suite is, first and foremost, a risk-mitigation measure.
Automated tests reduce the risk associated with changes to an existing codebase - most regressions and bugs are caught in the continuous integration pipeline and never reach users. The team is therefore empowered to iterate faster and release more often.

Tests act as documentation as well.
The test suite is often the best starting point when deep-diving in an unknown code base - it shows

you how the code is supposed to behave and what scenarios are considered relevant enough to have dedicated tests for.

"Write a test suite!" should definitely be on your to-do list if you want to make your project more welcoming to new contributors.

There are other positive side-effects often associated with good tests - modularity, decoupling. These are harder to quantify, as we have yet to agree as an industry on what "good code" looks like.

7.3.2 Why Don't We Write Tests?

Although there are compelling reasons to invest time and effort in writing a good test suite, reality is somewhat messier.

First, the development community did not always believe in the value of testing.
We can find examples of test-driven development throughout the history of the discipline, but it is only with the "Extreme Programming" (XP) book that the practice entered the mainstream debate - in 1999!

Paradigm shifts do not happen overnight - it took years for the test-driven approach to gain traction as a "best practice" within the industry.

If test-driven development has won the minds and hearts of developers, the battle with management is often still undergoing.
Good tests build technical leverage, but writing tests takes time. When a deadline is pressing, testing is often the first to be sacrificed.

As a consequence, most of the material you find around is either an introduction to testing or a guide on how to pitch its value to stakeholders.
There is very little about testing *at scale* - what happens if you stick to the book and keep writing tests as the codebase grows to tens of thousands of lines, with hundreds of test cases?

7.3.3 Test Code Is Still Code

All test suites start in the same way: an empty file, a world of possibilities.
You go in, you add the first test. Easy, done.
Then the second. Boom.
The third. You just had to copy a few lines from the first, all good.
The fourth...

After a while, test coverage starts to go down: new code is less thoroughly tested than the code you wrote at the very beginning of the project. Have you started to doubt the value of tests?
Absolutely not, tests are great!
Yet, you are writing fewer tests as the project moves forward.
It's because of friction - it got progressively more cumbersome to write new tests as the codebase evolved.

> Test code is still code.

It has to be modular, well-structured, sufficiently documented. It requires maintenance.
If we do not actively invest in the health of our test suite, it will rot over time.
Coverage goes down and soon enough we will find critical paths in our application code that are never exercised by automated tests.

You need to regularly step back to take a look at your test suite *as a whole*.
Time to look at ours, isn't it?

7.3.4 Our Test Suite

All our integration tests live within a single file, `tests/health_check.rs`:

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

// Ensure that the `tracing` stack is only initialised once using `once_cell`
static TRACING: Lazy<()> = Lazy::new(|| {
    // [...]
});

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
}

async fn spawn_app() -> TestApp {
    // [...]
}

pub async fn configure_database(config: &DatabaseSettings) -> PgPool {
    // [...]
}

#[tokio::test]
async fn health_check_works() {
    // [...]
}

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // [...]
}

#[tokio::test]
async fn subscribe_returns_a_400_when_data_is_missing() {
    // [...]
}

#[tokio::test]
async fn subscribe_returns_a_400_when_fields_are_present_but_invalid() {
    // [...]
}
```

7.3.5 Test Discovery

There is only one test dealing with our health check endpoint - `health_check_works`.
The other three tests are probing our `POST /subscriptions` endpoint while the rest of the code deals with shared setup steps (`spawn_app`, `TestApp`, `configure_database`, `TRACING`).

Why have we shoved everything in `tests/health_check.rs`?
Because it was convenient!
The setup functions were already there - it was easier to add another test case within the same file than figuring out how to share that code properly across multiple test modules.

Our main goal in this refactoring is *discoverability*:

- given an application endpoint, it should be easy to find the corresponding integration tests within the `tests` folder;
- when writing a test, it should be easy to find the relevant test helper functions.

We will focus on folder structure, but that is definitely not the only tool available when it comes to test discovery.
Test coverage tools can often tell you which tests triggered the execution of a certain line of application code.
You can rely on techniques such as coverage marks to create an obvious link between test and application code.

As always, a multi-pronged approach is likely to give you the best results as the complexity of your test suite increases.

7.3.6 One Test File, One Crate

Before we start moving things around, let's nail down a few facts about integration testing in Rust.
The `tests` folder is somewhat special - `cargo` knows to look into it searching for integration tests.

Each file within the `tests` folder gets compiled as its own crate.
We can check this out by running `cargo build --tests` and then looking under `target/debug/deps`:

```
# Build test code, without running tests
cargo build --tests
# Find all files with a name starting with `health_check`
ls target/debug/deps | grep health_check
```

```
health_check-fc23645bf877da35
health_check-fc23645bf877da35.d
```

The trailing hashes will likely be different on your machine, but there should be two entries starting with `health_check-*`.
What happens if you try to run it?

```
./target/debug/deps/health_check-fc23645bf877da35
```

```
running 4 tests
test health_check_works ... ok
test subscribe_returns_a_400_when_fields_are_present_but_invalid ... ok
test subscribe_returns_a_400_when_data_is_missing ... ok
test subscribe_returns_a_200_for_valid_form_data ... ok
test result: ok. 4 passed; finished in 0.44s
```

That's right, it runs our integration tests!
If we had five `*.rs` files under `tests`, we'd find five executables in `target/debug/deps`.

7.3.7 Sharing Test Helpers

If each integration test file is its own executable, how do we share test helpers functions?

The first option is to define a stand-alone module - e.g. `tests/helpers/mod.rs`[1].
You can add common functions in `mod.rs` (or define other sub-modules in there) and then refer to `helpers` in your test file (e.g. `tests/health_check.rs`) with:

```rust
//! tests/health_check.rs
// [...]
mod helpers;

// [...]
```

`helpers` is bundled in the `health_check` test executable as a sub-module and we get access to the functions it exposes in our test cases.
This approach works fairly well to start out, but it leads to annoying `function is never used` warnings down the line.
The issue is that `helpers` is bundled as a sub-module, it is not invoked as a third-party crate: `cargo` compiles each test executable in isolation and warns us if, for a specific test file, one or more public functions in `helpers` have never been invoked. This is bound to happen as your test suite grows - not all test files will use *all* your helper methods.

The second option takes full advantage of that each file under `tests` is its own executable - we can create sub-modules *scoped to a single test executable*!
Let's create an `api` folder under `tests`, with a single `main.rs` file inside:

```
tests/
  api/
    main.rs
  health_check.rs
```

First, we gain clarity: we are structuring `api` in the very same way we would structure a binary crate. Less magic - it builds on the same knowledge of the module system you built while working on application code.
If you run `cargo build --tests` you should be able to spot

```
Running target/debug/deps/api-0a1bfb817843fdcf

running 0 tests

test result: ok. 0 passed; finished in 0.00s
```

in the output - `cargo` compiled `api` as a test executable, looking for tests cases.
There is no need to define a `main` function in `main.rs` - the Rust test framework adds one for us behind the scenes[2].

[1] Refer to the test organization chapter in the Rust book for more details.
[2] You can actually override the default test framework and plug your own. Look at libtest-mimic as an example!

We can now add sub-modules in `main.rs`:

```rust
//! tests/api/main.rs

mod helpers;
mod health_check;
mod subscriptions;
```

Add three empty files - `tests/api/helpers.rs`, `tests/api/health_check.rs` and `tests/api/subscriptions.rs`.
Time to delete `tests/health_check.rs` and re-distribute its content:

```rust
//! tests/api/helpers.rs
use sqlx::{Connection, Executor, PgConnection, PgPool};
use std::net::TcpListener;
use uuid::Uuid;
use zero2prod::configuration::{get_configuration, DatabaseSettings};
use zero2prod::email_client::EmailClient;
use zero2prod::startup::run;
use zero2prod::telemetry::{get_subscriber, init_subscriber};

// Ensure that the `tracing` stack is only initialised once using `once_cell`
static TRACING: Lazy<()> = Lazy::new(|| {
    // [...]
});

pub struct TestApp {
    // [...]
}

// Public!
pub async fn spawn_app() -> TestApp {
    // [...]
}

// Not public anymore!
async fn configure_database(config: &DatabaseSettings) -> PgPool {
    // [...]
}
```

```rust
//! tests/api/health_check.rs
use crate::helpers::spawn_app;

#[tokio::test]
async fn health_check_works() {
    // [...]
}
```

```rust
//! tests/api/subscriptions.rs
use crate::helpers::spawn_app;

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // [...]
}

#[tokio::test]
```

```rust
async fn subscribe_returns_a_400_when_data_is_missing() {
    // [...]
}

#[tokio::test]
async fn subscribe_returns_a_400_when_fields_are_present_but_invalid() {
    // [...]
}
```

`cargo test` should succeed, with no warnings.
Congrats, you have broken down your test suite in smaller and more manageable modules!

There are a few positive side-effects to the new structure: - it is recursive.
If `tests/api/subscriptions.rs` grows too unwieldy, we can turn it into a module, with `tests/api/subscriptions/helpers.rs` holding subscription-specific test helpers and one or more test files focused on a specific flow or concern; - the implementation details of our helpers function are encapsulated.
It turns out that our tests only need to know about `spawn_app` and `TestApp` - no need to expose `configure_database` or `TRACING`, we can keep that complexity hidden away in the `helpers` module; - we have a single test binary.
If you have large test suite with a flat file structure, you'll soon be building tens of executable every time you run `cargo test`. While each executable is compiled in parallel, the linking phase is instead entirely sequential! Bundling all your test cases in a single executable reduces the time spent compiling your test suite in CI[3].

If you are running Linux, you might see errors like

```
thread 'actix-rt:worker' panicked at
'Can not create Runtime: Os { code: 24, kind: Other, message: "Too many open files" }',
```

when you run `cargo test` after the refactoring.
This is due to a limit enforced by the operating system on the maximum number of open file descriptors (including sockets) for each process - given that we are now running all tests as part of a single binary, we might be exceeding it. The limit is usually set to 1024, but you can raise it with `ulimit -n X` (e.g. `ulimit -n 10000`) to resolve the issue.

7.3.8 Sharing Startup Logic

Now that we have reworked the layout of our test suite it's time to zoom in on the test logic itself.
We will start with `spawn_app`:

```rust
//! tests/api/helpers.rs
// [...]

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
}

pub async fn spawn_app() -> TestApp {
    Lazy::force(&TRACING);
```

[3] See this article as an example with some numbers (1.9x speedup!). You should always benchmark the approach on your specific codebase before committing.

```rust
    let listener = TcpListener::bind("127.0.0.1:0").expect("Failed to bind random port");
    let port = listener.local_addr().unwrap().port();
    let address = format!("http://127.0.0.1:{}", port);

    let mut configuration = get_configuration().expect("Failed to read configuration.");
    configuration.database.database_name = Uuid::new_v4().to_string();
    let connection_pool = configure_database(&configuration.database).await;

    let sender_email = configuration
        .email_client
        .sender()
        .expect("Invalid sender email address.");
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
    );

    let server = run(listener, connection_pool.clone(), email_client)
        .expect("Failed to bind address");
    let _ = tokio::spawn(server);
    TestApp {
        address,
        db_pool: connection_pool,
    }
}

// [...]
```

Most of the code we have here is extremely similar to what we find in our `main` entrypoint:

```rust
//! src/main.rs
use sqlx::postgres::PgPoolOptions;
use std::net::TcpListener;
use zero2prod::configuration::get_configuration;
use zero2prod::email_client::EmailClient;
use zero2prod::startup::run;
use zero2prod::telemetry::{get_subscriber, init_subscriber};

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into(), std::io::stdout);
    init_subscriber(subscriber);

    let configuration = get_configuration().expect("Failed to read configuration.");
    let connection_pool = PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        .connect_lazy_with(configuration.database.with_db());

    let sender_email = configuration
        .email_client
        .sender()
        .expect("Invalid sender email address.");
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
    );

    let address = format!(
        "{}:{}",
        configuration.application.host, configuration.application.port
    );
    let listener = TcpListener::bind(address)?;
    run(listener, connection_pool, email_client)?.await?;
    Ok(())
}
```

Every time we add a dependency or modify the server constructor, we have at least two places to modify - we have recently gone through the motions with `EmailClient`. It's mildly annoying.
More importantly though, the startup logic in our application code is never tested.
As the codebase evolves, they might start to diverge subtly, leading to different behaviour in our tests compared to our production environment.

We will first extract the logic out of `main` and then figure out what hooks we need to leverage the same code paths in our test code.

7.3.8.1 Extracting Our Startup Code

From a structural perspective, our startup logic is a function taking `Settings` as input and returning an instance of our application as output.
It follows that our `main` function should look like this:

```
//! src/main.rs
use zero2prod::configuration::get_configuration;
use zero2prod::startup::build;
use zero2prod::telemetry::{get_subscriber, init_subscriber};

#[tokio::main]
async fn main() -> std::io::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into(), std::io::stdout);
    init_subscriber(subscriber);

    let configuration = get_configuration().expect("Failed to read configuration.");
    let server = build(configuration).await?;
    server.await?;
    Ok(())
}
```

We first perform some binary-specific logic (i.e. telemetry initialisation), then we build a set of configuration values from the supported sources (files + environment variables) and use it to spin up an application. Linear.
Let's define that `build` function then:

```
//! src/startup.rs
// [...]
// New imports!
use crate::configuration::Settings;
use sqlx::postgres::PgPoolOptions;

pub async fn build(configuration: Settings) -> Result<Server, std::io::Error> {
```

```rust
    let connection_pool = PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        .connect_lazy_with(configuration.database.with_db());

    let sender_email = configuration
        .email_client
        .sender()
        .expect("Invalid sender email address.");
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
    );

    let address = format!(
        "{}:{}",
        configuration.application.host, configuration.application.port
    );
    let listener = TcpListener::bind(address)?;
    run(listener, connection_pool, email_client)
}


pub fn run(
    listener: TcpListener,
    db_pool: PgPool,
    email_client: EmailClient,
) -> Result<Server, std::io::Error> {
    // [...]
}
```

Nothing too surprising - we have just moved around the code that was previously living in `main`. Let's make it test-friendly now!

7.3.8.2 Testing Hooks In Our Startup Logic

Let's look at our `spawn_app` function again:

```rust
//! tests/api/helpers.rs
// [...]
use zero2prod::startup::build;
// [...]

pub async fn spawn_app() -> TestApp {
    // The first time `initialize` is invoked the code in `TRACING` is executed.
    // All other invocations will instead skip execution.
    Lazy::force(&TRACING);

    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 address = format!("http://127.0.0.1:{}", port);

    let mut configuration = get_configuration().expect("Failed to read configuration.");
    configuration.database.database_name = Uuid::new_v4().to_string();
    let connection_pool = configure_database(&configuration.database).await;
    let sender_email = configuration
        .email_client
        .sender()
        .expect("Invalid sender email address.");
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
    );

    let server = run(listener, connection_pool.clone(), email_client)
        .expect("Failed to bind address");
    let _ = tokio::spawn(server);
    TestApp {
        address,
        db_pool: connection_pool,
    }
}

// [...]
```

At a high-level, we have the following phases:

- Execute test-specific setup (i.e. initialise a `tracing` subscriber);
- Randomise the configuration to ensure tests do not interfere with each other (i.e. a different logical database for each test case);
- Initialise external resources (e.g. create and migrate the database!);
- Build the application;
- Launch the application as a background task and return a set of resources to interact with it.

Can we just throw `build` in there and call it a day?
Not really, but let's try to see where it falls short:

```rust
//! tests/api/helpers.rs
// [...]
// New import!
use zero2prod::startup::build;

pub async fn spawn_app() -> TestApp {
    Lazy::force(&TRACING);

    // Randomise configuration to ensure test isolation
    let configuration = {
        let mut c = get_configuration().expect("Failed to read configuration.");
        // Use a different database for each test case
        c.database.database_name = Uuid::new_v4().to_string();
        // Use a random OS port
        c.application.port = 0;
        c
    };

    // Create and migrate the database
    configure_database(&configuration.database).await;
    // Launch the application as a background task
    let server = build(configuration).await.expect("Failed to build application.");
    let _ = tokio::spawn(server);

    TestApp {
        // How do we get these?
        address: todo!(),
        db_pool: todo!()
    }
}

// [...]
```

It *almost* works - the approach falls short at the very end: we have no way to retrieve the random address assigned by the OS to the application and we don't really know how to build a connection pool to the database, needed to perform assertions on side-effects impacting the persisted state.

Let's deal with the connection pool first: we can extract the initialisation logic from `build` into a stand-alone function and invoke it twice.

```rust
//! src/startup.rs
// [...]
use crate::configuration::DatabaseSettings;

// We are taking a reference now!
pub async fn build(configuration: &Settings) -> Result<Server, std::io::Error> {
    let connection_pool = get_connection_pool(&configuration.database);
    // [...]
}

pub fn get_connection_pool(
    configuration: &DatabaseSettings
) -> PgPool {
    PgPoolOptions::new()
        .connect_timeout(std::time::Duration::from_secs(2))
        .connect_lazy_with(configuration.with_db())
}
```

```rust
//! tests/api/helpers.rs
// [...]
use zero2prod::startup::{build, get_connection_pool};
// [...]

pub async fn spawn_app() -> TestApp {
    // Notice the .clone!
    let server = build(configuration.clone())
        .await
        .expect("Failed to build application.");
    // [...]
    TestApp {
        address: todo!(),
        db_pool: get_connection_pool(&configuration.database),
    }
}

// [...]
```

You'll have to add a `#[derive(Clone)]` to all the structs in `src/configuration.rs` to make the compiler happy, but we are done with the database connection pool.

How do we get the application address instead?
`actix_web::dev::Server`, the type returned by `build`, does not allow us to retrieve the application port.
We need to do a bit more legwork in our application code - we will wrap `actix_web::dev::Server` in a new type that holds on to the information we want.

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

// A new type to hold the newly built server and its port
pub struct Application {
    port: u16,
    server: Server,
}

impl Application {
    // We have converted the `build` function into a constructor for
    // `Application`.
    pub async fn build(configuration: Settings) -> Result<Self, std::io::Error> {
        let connection_pool = get_connection_pool(&configuration.database);

        let sender_email = configuration
            .email_client
            .sender()
            .expect("Invalid sender email address.");
        let email_client = EmailClient::new(
            configuration.email_client.base_url,
            sender_email,
            configuration.email_client.authorization_token,
        );

        let address = format!(
            "{}:{}",
            configuration.application.host, configuration.application.port
        );
        let listener = TcpListener::bind(&address)?;
        let port = listener.local_addr().unwrap().port();
        let server = run(listener, connection_pool, email_client)?;

        // We "save" the bound port in one of `Application`'s fields
        Ok(Self { port, server })
    }

    pub fn port(&self) -> u16 {
        self.port
    }

    // A more expressive name that makes it clear that
    // this function only returns when the application is stopped.
    pub async fn run_until_stopped(self) -> Result<(), std::io::Error> {
        self.server.await
    }
}
```

```
// [...]
```

```rust
//! tests/api/helpers.rs
// [...]
// New import!
use zero2prod::startup::Application;

pub async fn spawn_app() -> TestApp {
    // [...]

    let application = Application::build(configuration.clone())
        .await
        .expect("Failed to build application.");
    // Get the port before spawning the application
    let address = format!("http://127.0.0.1:{}", application.port());
    let _ = tokio::spawn(application.run_until_stopped());

    TestApp {
        address,
        db_pool: get_connection_pool(&configuration.database),
    }
}

// [...]
```

```rust
//! src/main.rs
// [...]
// New import!
use zero2prod::startup::Application;

#[tokio::main]
async fn main() -> std::io::Result<()> {
    // [...]
    let application = Application::build(configuration).await?;
    application.run_until_stopped().await?;
    Ok(())
}
```

It's done - run `cargo test` if you want to double-check!

7.3.9 Build An API Client

All of our integration tests are black-box: we launch our application at the beginning of each test and interact with it using an HTTP client (i.e. `reqwest`).
As we write tests, we necessarily end up implementing a client for our API.

That's great!
It gives us a prime opportunity to see what it feels like to interact with the API as a user.
We just need to be careful not to spread the client logic all over the test suite - when the API changes, we don't want to go through tens of tests to remove a trailing `s` from the path of an endpoint.

Let's look at our subscriptions tests:

```rust
//! tests/api/subscriptions.rs
use crate::helpers::spawn_app;
```

```rust
#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app = spawn_app().await;
    let client = reqwest::Client::new();
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    // Act
    let response = client
        .post(&format!("{}/subscriptions", &app.address))
        .header("Content-Type", "application/x-www-form-urlencoded")
        .body(body)
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(200, response.status().as_u16());

    let saved = sqlx::query!("SELECT email, name FROM subscriptions",)
        .fetch_one(&app.db_pool)
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
}

#[tokio::test]
async fn subscribe_returns_a_400_when_data_is_missing() {
    // Arrange
    let app = spawn_app().await;
    let client = reqwest::Client::new();
    let test_cases = vec![
        ("name=le%20guin", "missing the email"),
        ("email=ursula_le_guin%40gmail.com", "missing the name"),
        ("", "missing both name and email"),
    ];

    for (invalid_body, error_message) in test_cases {
        // Act
        let response = client
            .post(&format!("{}/subscriptions", &app.address))
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(invalid_body)
            .send()
            .await
            .expect("Failed to execute request.");

        // Assert
        assert_eq!(
            400,
            response.status().as_u16(),
            // Additional customised error message on test failure
            "The API did not fail with 400 Bad Request when the payload was {}.",
            error_message
        );
    }
}

#[tokio::test]
async fn subscribe_returns_a_400_when_fields_are_present_but_invalid() {
    // Arrange
    let app = spawn_app().await;
    let client = reqwest::Client::new();
    let test_cases = vec![
        ("name=&email=ursula_le_guin%40gmail.com", "empty name"),
        ("name=Ursula&email=", "empty email"),
        ("name=Ursula&email=definitely-not-an-email", "invalid email"),
    ];

    for (body, description) in test_cases {
        // Act
        let response = client
            .post(&format!("{}/subscriptions", &app.address))
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(body)
            .send()
            .await
            .expect("Failed to execute request.");

        // Assert
        assert_eq!(
            400,
            response.status().as_u16(),
            "The API did not return a 400 Bad Request when the payload was {}.",
            description
        );
    }
}
```

We have the same calling code in each test - we should pull it out and add a helper method to our `TestApp` struct:

```rust
//! tests/api/helpers.rs
// [...]

pub struct TestApp {
    // [...]
}

impl TestApp {
    pub async fn post_subscriptions(&self, body: String) -> reqwest::Response {
        reqwest::Client::new()
            .post(&format!("{}/subscriptions", &self.address))
            .header("Content-Type", "application/x-www-form-urlencoded")
            .body(body)
            .send()
            .await
            .expect("Failed to execute request.")
    }
}
```

```rust
// [...]
```

```rust
//! tests/api/subscriptions.rs
use crate::helpers::spawn_app;

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // [...]
    // Act
    let response = app.post_subscriptions(body.into()).await;
    // [...]
}

#[tokio::test]
async fn subscribe_returns_a_400_when_data_is_missing() {
    // [...]
    for (invalid_body, error_message) in test_cases {
        let response = app.post_subscriptions(invalid_body.into()).await;
        // [...]
    }
}

#[tokio::test]
async fn subscribe_returns_a_400_when_fields_are_present_but_invalid() {
    // [...]
    for (body, description) in test_cases {
        let response = app.post_subscriptions(body.into()).await;
        // [...]
    }
}
```

We could add another method for the health check endpoint, but it's only used once - there is no need right now.

7.3.10 Summary

We started with a single file test suite, we finished with a modular test suite and a robust set of helpers.
Just like application code, test code is never finished: we will have to keep working on it as the project evolves, but we have laid down solid foundations to keep moving forward without losing momentum.
We are now ready to tackle the remaining pieces of functionality needed to dispatch a confirmation email.

7.4 Refocus

Time to go back to the plan we drafted at the beginning of the chapter:

- write a module to send an email;
- adapt the logic of our existing `POST /subscriptions` request handler to match the new requirements;
- write a `GET /subscriptions/confirm` request handler from scratch.

The first item is done, time to move on to the remaining two on the list.
We had a sketch of how the two handlers should work:

`POST /subscriptions` will:

- add the subscriber details to the database in the `subscriptions` table, with `status` equal to `pending_confirmation`;
- generate a (unique) `subscription_token`;
- store `subscription_token` in our database against the subscriber `id` in a `subscription_tokens` table;
- send an email to the new subscriber containing a link structured as `https://<our-api-domain>/subscriptions/confirm?token=<subscription_token>`;
- return a `200 OK`.

Once they click on the link, a browser tab will open up and a `GET` request will be fired to our `GET /subscriptions/confirm` endpoint. The request handler will:

- retrieve `subscription_token` from the query parameters;
- retrieve the subscriber `id` associated with `subscription_token` from the `subscription_tokens` table;
- update the subscriber status from `pending_confirmation` to `active` in the `subscriptions` table;
- return a `200 OK`.

This gives us a fairly precise picture of how the application is going to work once we are done with the implementation.
It does not help us much to figure out **how to get there**.

Where should we start from?
Should we immediately tackle the changes to `/subscriptions`?
Should we get `/subscriptions/confirm` out of the way?

We need to find an implementation route that can be rolled out with **zero downtime**.

7.5 Zero Downtime Deployments

7.5.1 Reliability

In Chapter 5 we deployed our application to a public cloud provider.
It is live: we are not sending out newsletter issues yet, but people can subscribe while we figure that part out.

Once an application is serving production traffic, we need to make sure it is **reliable**.

Reliable means different things in different contexts. If you are selling a data storage solution, for example, it should not lose (or corrupt!) customers' data.

In a commercial setting, the definition of reliability for your application will often be encoded in a Service Level Agreement (SLA).
An SLA is a contractual obligation: you guarantee a certain level of reliability and commit to compensate your customers (usually with discounts or credits) if your service fails to live up to the expectations.

If you are selling access to an API, for example, you will usually have something related to **availability** - e.g. the API should successfully respond to at least 99.99% of well-formed incoming requests, often referred to as "four nines of availability".
Phrased differently (and assuming a uniform distribution of incoming requests over time), you can only afford up to 52 minutes of downtime over a whole year. Achieving four nines of availability is tough.

There is no silver bullet to build a highly available solution: it requires work from the application layer all the way down to the infrastructure layer.
One thing is certain, though: if you want to operate a highly available service, you should master **zero downtime deployments** - users should be able to use the service before, during and after the rollout of a new version of the application to production.
This is even more important if you are practising continuous deployment: you cannot release multiple times a day if every release triggers a small outage.

7.5.2 Deployment Strategies

7.5.2.1 Naive Deployment

Before diving deeper into zero downtime deployments, let's have a look at the "naive" approach.
Version `A` of our service is running in production and we want to roll out version `B`:

- We switch off all instances of version `A` running the cluster;
- We spin up new instances of our application running version `B`;
- We start serving traffic using version `B`.

There is a non-zero amount of time where there is no application running in the cluster able to serve user traffic - we are experiencing downtime!

To do better we need to take a closer look at how our infrastructure is set up.

7.5.2.2 Load Balancers

We have multiple copies[4] of our application running behind a **load balancer**.
Each replica of our application is registered with the load balancer as a **backend**.
Every time somebody sends a request to our API, they hit our load balancer which is then in charge of choosing one of the available backends to fulfill the incoming request.

Load balancers usually support adding (and removing) backends **dynamically**.
This enables a few interesting patterns.

[4] You might recall that we set the number of replicas to 1 in chapter 5 to reduce the bill while experimenting. Even if you are running a single replica there is load balancer between your users and your application. Deployments are still performed using a rolling update strategy.

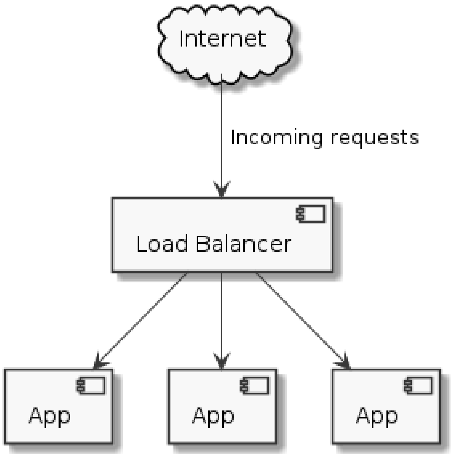


Figure 7.2: Load balancer

7.5.2.2.1 Horizontal Scaling We can add more capacity when experiencing a traffic spike by spinning up more replicas of our application (i.e. horizontal scaling).
It helps to spread the load until the work expected of a single instance becomes manageable.

We will get back to this topic later in the book when discussing metrics and autoscaling.

7.5.2.2.2 Health Checks We can ask the load balancer to keep an eye on the **health** of the registered backends.
Oversimplifying, health checking can be:

- Passive - the load balancer looks at the distribution of status codes/latency for each backend to determine if they are healthy or not;
- Active - the load balancer is configured to send a health check request to each backend on a schedule. If a backend fails to respond with a success status code for a long enough time period it is marked as unhealthy and removed.

This is a critical capability to achieve **self-healing** in a cloud-native environment: the platform can detect if an application is not behaving as expected and automatically remove it from the list of available backends to mitigate or nullify the impact on users[5].

7.5.2.3 Rolling Update Deployments

We can leverage our load balancer to perform zero downtime deployments.

Let's look at a snapshot of our production environments: we have three replicas of version `A` of our application registered as backends for our load balancer.
We want to deploy version `B`.

We start by spinning up one replica of version `B` of our application.
When the application is ready to serve traffic (i.e. a few health check requests have succeeded) we register it as a backend with our load balancer.

[5] This is true as long as the platform is also capable of automatically provisioning a new replica of the application when the number of healthy instances falls below a pre-determined threshold.

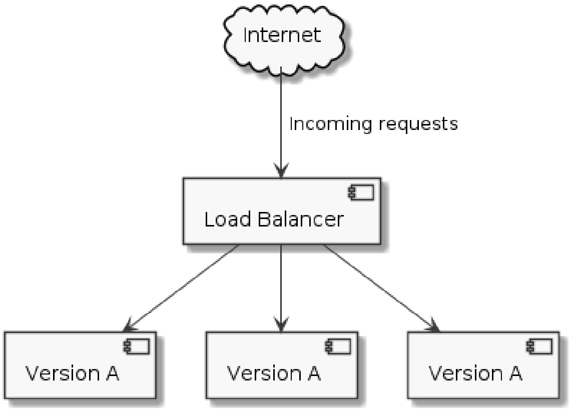


Figure 7.3: System before the roll out begins.

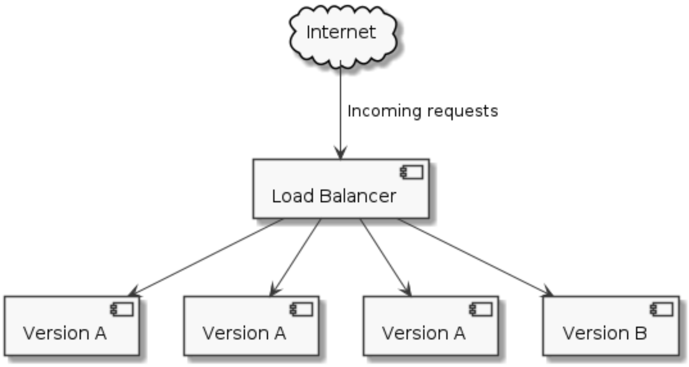


Figure 7.4: One operational instance of version B.

We have four replicas of our application now: 3 running version `A`, 1 running version `B`. **All four** are serving live traffic.
If all is well, we switch off one of the replicas running version `A`.

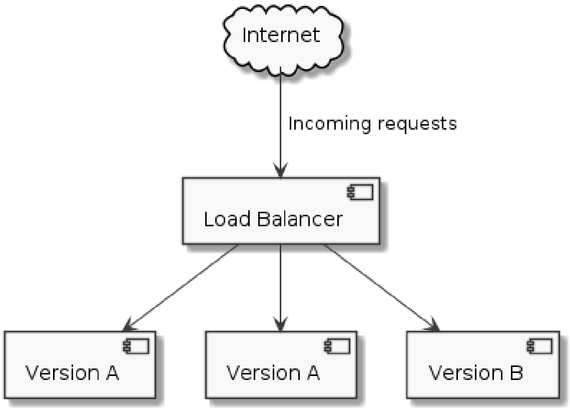


Figure 7.5: One instance of version A has been decommissioned.

We follow the same process to replace all replicas running version `A` until all registered backends are running version `B`.

This deployment strategy is called **rolling update**: we run the old and the new version of the application side by side, serving live traffic with both.
Throughout the process we always have three or more healthy backends: users should not experience any kind of service degradation (assuming version B is not buggy).

7.5.2.4 Digital Ocean App Platform

We are running our application on Digital Ocean App Platform.
Their documentation boasts of offering zero downtime deployments out of the box, but they do not provide details on how it is achieved.

A few experiments confirmed that they are indeed relying on a rolling update deployment strategy.

A rolling update is not the only possible strategy for a zero downtime deployment - blue-green and canary deployments are equally popular variations over the same underlying principles.
Choose the most appropriate solution for your application based on the capabilities offered by your platform and your requirements.

7.6 Database Migrations

7.6.1 State Is Kept Outside The Application

Load balancing relies on a strong assumption: no matter which backend is used to serve an incoming request, the outcome will be the same.

This is something we discussed already in Chapter 3: to ensure high availability in a fault-prone environment, cloud-native applications are **stateless** - they delegate all persistence concerns to external systems (i.e. databases).

That's why load balancing works: all backends are talking to the same database to query and manipulate the same **state**.

Think of a database as a single gigantic global variable. Continuously accessed and mutated by all replicas of our application.
State is hard.

7.6.2 Deployments And Migrations

During a rolling update deployment, the old and the new version of the application are both serving live traffic, side by side.
From a different perspective: the old and the new version of the application are using the **same database** at the **same time**.

To avoid downtime, we need a database schema that is understood by both versions.
This is not an issue for most of our deployments, but it is a serious constraint when we need to evolve the schema.

Let's circle back to the job we set out to do, confirmation emails.
To move forward with the implementation strategy we identified, we need to evolve our database schema as follows:

- add a new table, `subscription_tokens`;
- add a new mandatory column, `status`, to the existing `subscriptions` table.

Let's go over the possible scenarios to convince ourselves that we cannot possibly deploy confirmation emails all at once without incurring downtime.

We could first migrate the database and then deploy the new version.
This implies that the current version is running against the migrated database for some time: our current implementation of `POST /subscriptions` does not know about `status` and it tries to insert new rows into `subscriptions` without populating it. Given that `status` is constrained to be `NOT NULL` (i.e. it's mandatory), all inserts would fail - we would not be able to accept new subscribers until the new version of the application is deployed.
Not good.

We could first deploy the new version and then migrate the database.
We get the opposite scenario: the new version of the application is running against the old database schema. When `POST /subscriptions` is called, it tries to insert a row into `subscriptions` with a `status` field that does not exist - all inserts fail and we cannot accept new subscribers until the database is migrated.
Once again, not good.

7.6.3 Multi-step Migrations

A big bang release won't cut it - we need to get there in multiple, smaller steps.
The pattern is somewhat similar to what we see in test-driven development: we don't change code and tests at the same time - one of the two needs to stay still while the other changes.

The same applies to database migrations and deployments: if we want to evolve the database schema we cannot change the application behaviour at the same time.

Think of it as a database refactoring: we are laying down the foundations in order to build the behaviour we need later on.

7.6.4 A New Mandatory Column

Let's start by looking at the `status` column.

7.6.4.1 Step 1: Add As Optional

We start by keeping the application code stable.
On the database side, we generate a new migration script:

```
sqlx migrate add add_status_to_subscriptions
```

```
Creating migrations/20210307181858_add_status_to_subscriptions.sql
```

We can now edit the migration script to add `status` as an **optional** column to `subscriptions`:

```sql
ALTER TABLE subscriptions ADD COLUMN status TEXT NULL;
```

Run the migration against your local database (`SKIP_DOCKER=true ./scripts/init_db.sh`): we can now run our test suite to make sure that the code works as is even against the new database schema.

It should pass: go ahead and migrate the production database.

7.6.4.2 Step 2: Start Using The New Column

`status` now exists: we can start using it!
To be precise, we can start writing to it: every time a new subscriber is inserted, we will set status to `confirmed`.

We just need to change our insertion query from

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber([...]) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"INSERT INTO subscriptions (id, email, name, subscribed_at)
        VALUES ($1, $2, $3, $4)"#,
        // [...]
    )
    // [...]
}
```

to

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber([...]) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"INSERT INTO subscriptions (id, email, name, subscribed_at, status)
        VALUES ($1, $2, $3, $4, 'confirmed')"#,
        // [...]
    )
    // [...]
}
```

Tests should pass - deploy the new version of the application to production.

7.6.4.3 Step 3: Backfill And Mark As `NOT NULL`

The latest version of the application ensures that `status` is populated for all new subscribers.
To mark `status` as `NOT NULL` we just need to backfill the value for historical records: we'll then be free to alter the column.

Let's generate a new migration script:

```
sqlx migrate add make_status_not_null_in_subscriptions
```

```
Creating migrations/20210307184428_make_status_not_null_in_subscriptions.sql
```

The SQL migration looks like this:

```sql
-- We wrap the whole migration in a transaction to make sure
-- it succeeds or fails atomically. We will discuss SQL transactions
-- in more details towards the end of this chapter!
-- `sqlx` does not do it automatically for us.
BEGIN;
    -- Backfill `status` for historical entries
    UPDATE subscriptions
        SET status = 'confirmed'
        WHERE status IS NULL;
    -- Make `status` mandatory
    ALTER TABLE subscriptions ALTER COLUMN status SET NOT NULL;
COMMIT;
```

We can migrate our local database, run our test suite and then deploy our production database.
We made it, we added `status` as a new mandatory column!

7.6.5 A New Table

What about `subscription_tokens`? Do we need three steps there as well?

No, it is much simpler: we add the new table in a migration while the application keeps ignoring it.
We can then deploy a new version of the application that uses it to enable confirmation emails.

Let's generate a new migration script:

```
sqlx migrate add create_subscription_tokens_table
```

```
Creating migrations/20210307185410_create_subscription_tokens_table.sql
```

The migration is similar to the very first one we wrote to add `subscriptions`:

```sql
-- Create Subscription Tokens Table
CREATE TABLE subscription_tokens(
   subscription_token TEXT NOT NULL,
   subscriber_id uuid NOT NULL
      REFERENCES subscriptions (id),
   PRIMARY KEY (subscription_token)
);
```

Pay attention to the details here: the `subscriber_id` column in `subscription_tokens` is a **foreign key**.
For each row in `subscription_tokens` there must exist a row in `subscriptions` whose `id` field has the same value of `subscriber_id`, otherwise the insertion fails. This guarantees that all tokens are attached to a legitimate subscriber.

Migrate the production database again - we are done!

7.7 Sending A Confirmation Email

It took us a while, but the groundwork is done: our production database is ready to accommodate the new feature we want to build, confirmation emails.
Time to focus on the application code.

We will build the whole feature in a proper test-driven fashion: small steps in a tight red-green-refactor loop. Get ready!

7.7.1 A Static Email

We will start simple: we will test that `POST /subscriptions` is sending out an email.
We will not be looking, at this stage, at the body of the email - in particular, we will not check that it contains a confirmation link.

7.7.1.1 Red test

To write this test we need to enhance our `TestApp`.

It currently holds our application and a handle to a pool of connections to the database:

```
//! tests/api/helpers.rs
// [...]

pub struct TestApp {
  pub address: String,
  pub db_pool: PgPool,
}
```

We need to spin up a mock server to stand in for Postmark's API and intercept outgoing requests, just like we did when we built the email client.

Let's edit `spawn_app` accordingly:

```
//! tests/api/helpers.rs

// New import!
use wiremock::MockServer;
// [...]

pub struct TestApp {
    pub address: String,
    pub db_pool: PgPool,
```

```rust
    // New field!
    pub email_server: MockServer,
}

pub async fn spawn_app() -> TestApp {
    // [...]
    // Launch a mock server to stand in for Postmark's API
    let email_server = MockServer::start().await;

    // Randomise configuration to ensure test isolation
    let configuration = {
        let mut c = get_configuration().expect("Failed to read configuration.");
        // [...]
        // Use the mock server as email API
        c.email_client.base_url = email_server.uri();
        c
    };

    // [...]

    TestApp {
        // [...],
        email_server,
    }
}
```

We can now write the new test case:

```rust
//! tests/api/subscriptions.rs
// New imports!
use wiremock::matchers::{method, path};
use wiremock::{Mock, ResponseTemplate};
// [...]

#[tokio::test]
async fn subscribe_sends_a_confirmation_email_for_valid_data() {
  // Arrange
  let app = spawn_app().await;
  let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

  Mock::given(path("/email"))
          .and(method("POST"))
          .respond_with(ResponseTemplate::new(200))
          .expect(1)
          .mount(&app.email_server)
          .await;

  // Act
  app.post_subscriptions(body.into()).await;

  // Assert
  // Mock asserts on drop
}
```

The test, as expected, fails:

```
failures:

---- subscriptions::subscribe_sends_a_confirmation_email_for_valid_data stdout ----
thread 'subscriptions::subscribe_sends_a_confirmation_email_for_valid_data' panicked at
'Verifications failed:
- Mock #0.
        Expected range of matching incoming requests: == 1
        Number of matched incoming requests: 0'
```

Notice that, on failure, `wiremock` gives us a detailed breakdown of what happened: we expected an incoming request, we received none.
Let's fix that.

7.7.1.2 Green test

Our handler looks like this right now:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(form: web::Form<FormData>, pool: web::Data<PgPool>) -> HttpResponse {
    let new_subscriber = match form.0.try_into() {
        Ok(form) => form,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    match insert_subscriber(&pool, &new_subscriber).await {
        Ok(_) => HttpResponse::Ok().finish(),
        Err(_) => HttpResponse::InternalServerError().finish(),
    }
}
```

To send an email we need to get our hands on an instance of `EmailClient`.
As part of the work we did when writing the module, we also registered it in the application context:

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

fn run([...]) -> Result<Server, std::io::Error> {
    // [...]
    let email_client = Data::new(email_client);
    let server = HttpServer::new(move || {
        App::new()
            .wrap(TracingLogger::default())
            // [...]
            // Here!
            .app_data(email_client.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

We can therefore access it in our handler using `web::Data`, just like we did for `pool`:

```rust
//! src/routes/subscriptions.rs
// New import!
use crate::email_client::EmailClient;
// [...]

#[tracing::instrument(
    name = "Adding a new subscriber",
    skip(form, pool, email_client),
    fields(
        subscriber_email = %form.email,
        subscriber_name= %form.name
    )
)]
pub async fn subscribe(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
    // Get the email client from the app context
    email_client: web::Data<EmailClient>,
) -> HttpResponse {
    // [...]
    if insert_subscriber(&pool, &new_subscriber).await.is_err() {
        return HttpResponse::InternalServerError().finish();
    }
    // Send a (useless) email to the new subscriber.
    // We are ignoring email delivery errors for now.
    if email_client
        .send_email(
            new_subscriber.email,
            "Welcome!",
            "Welcome to our newsletter!",
            "Welcome to our newsletter!",
        )
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    HttpResponse::Ok().finish()
}
```

`subscribe_sends_a_confirmation_email_for_valid_data` now passes, but `subscribe_returns_a_200_for_valid_form_data` fails:

```
thread 'subscriptions::subscribe_returns_a_200_for_valid_form_data' panicked at
'assertion failed: `(left == right)`
  left: `200`,
 right: `500`'
```

It is trying to send an email but it is failing because we haven't setup a mock in that test. Let's fix it:

```rust
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    // New section!
    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    // Act
    let response = app.post_subscriptions(body.into()).await;

    // Assert
    assert_eq!(200, response.status().as_u16());

    // [...]
}
```

All good, the test passes now.
There is not much to refactor at the moment, let's press forward.

7.7.2 A Static Confirmation Link

Let's raise the bar a bit - we will scan the body of the email to retrieve a confirmation link.

7.7.2.1 Red Test

We don't care (yet) about the link being dynamic or actually meaningful - we just want to be sure that there is *something* in the body that looks like a link.
We should also have the same link in both the plain text and the HTML version of the email body.

How do we get the body of a request intercepted by `wiremock::MockServer`?
We can use its `received_requests` method - it returns a vector of all the requests intercepted by the server as long as request recording was enabled (the default).

```rust
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_sends_a_confirmation_email_with_a_link() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        // We are not setting an expectation here anymore
        // The test is focused on another aspect of the app
        // behaviour.
        .mount(&app.email_server)
        .await;

    // Act
    app.post_subscriptions(body.into()).await;

    // Assert
    // Get the first intercepted request
    let email_request = &app.email_server.received_requests().await.unwrap()[0];
    // Parse the body as JSON, starting from raw bytes
    let body: serde_json::Value = serde_json::from_slice(&email_request.body).unwrap();
}
```

We now need to extract links out of it.
The most obvious way forward would be a regular expression. Let's face it though: regexes are a messy business and it takes a while to get them right.
Once again, we can leverage the work done by the larger Rust ecosystem - let's add `linkify` as a development dependency:

```
#! Cargo.toml
# [...]
[dev-dependencies]
linkify = "0.8"
# [...]
```

We can use `linkify` to scan text and return an iterator of extracted links.

```
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_sends_a_confirmation_email_with_a_link() {
    // [...]
    let body: serde_json::Value = serde_json::from_slice(&email_request.body).unwrap();

    // Extract the link from one of the request fields.
    let get_link = |s: &str| {
        let links: Vec<_> = linkify::LinkFinder::new()
            .links(s)
            .filter(|l| *l.kind() == linkify::LinkKind::Url)
            .collect();
            assert_eq!(links.len(), 1);
        links[0].as_str().to_owned()
    };

    let html_link = get_link(&body["HtmlBody"].as_str().unwrap());
    let text_link = get_link(&body["TextBody"].as_str().unwrap());
    // The two links should be identical
    assert_eq!(html_link, text_link);
}
```

If we run the test suite, we should see the new test case failing:

```
failures:

thread 'subscriptions::subscribe_sends_a_confirmation_email_with_a_link'
panicked at 'assertion failed: `(left == right)`
  left: `0`,
 right: `1`', tests/api/subscriptions.rs:71:9
```

7.7.2.2 Green Test

We need to tweak our request handler again to satisfy the new test case:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(/* */) -> HttpResponse {
    // [...]
    let confirmation_link =
        "https://my-api.com/subscriptions/confirm";
    if email_client
        .send_email(
            new_subscriber.email,
            "Welcome!",
            &format!(
                "Welcome to our newsletter!<br />\
                Click <a href=\"{}\">here</a> to confirm your subscription.",
                confirmation_link
            ),
            &format!(
                "Welcome to our newsletter!\nVisit {} to confirm your subscription.",
                confirmation_link
            ),
        )
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    HttpResponse::Ok().finish()
}
```

The test should pass straight away.

7.7.2.3 Refactor

Our request handler is getting a bit busy - there is a lot of code dealing with our confirmation email now.
Let's extract it into a separate function:

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn subscribe(/* */) -> HttpResponse {
    let new_subscriber = match form.0.try_into() {
        Ok(form) => form,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    if insert_subscriber(&pool, &new_subscriber).await.is_err() {
        return HttpResponse::InternalServerError().finish();
    }
    if send_confirmation_email(&email_client, new_subscriber)
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    HttpResponse::Ok().finish()
}

#[tracing::instrument(
    name = "Send a confirmation email to a new subscriber",
    skip(email_client, new_subscriber)
)]
pub async fn send_confirmation_email(
    email_client: &EmailClient,
    new_subscriber: NewSubscriber,
) -> Result<(), reqwest::Error> {
    let confirmation_link = "https://my-api.com/subscriptions/confirm";
    let plain_body = format!(
        "Welcome to our newsletter!\nVisit {} to confirm your subscription.",
        confirmation_link
    );
    let html_body = format!(
        "Welcome to our newsletter!<br />\
        Click <a href=\"{}\">here</a> to confirm your subscription.",
      confirmation_link
    );
    email_client
        .send_email(
            new_subscriber.email,
            "Welcome!",
            &html_body,
            &plain_body,
        )
       .await
}
```

`subscribe` is once again focused on the overall flow, without bothering with details of any of its steps.

7.7.3 Pending Confirmation

Let's look at the status for a new subscriber now.
We are currently setting their status to `confirmed` in `POST /subscriptions`, while it should be `pending_confirmation` until they click on the confirmation link.

Time to fix it.

7.7.3.1 Red test

We can start by having a second look at our first "happy path" test:

```
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    // Act
    let response = app.post_subscriptions(body.into()).await;

    // Assert
    assert_eq!(200, response.status().as_u16());

    let saved = sqlx::query!("SELECT email, name FROM subscriptions",)
        .fetch_one(&app.db_pool)
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
}
```

The name is a bit of a lie - it is checking the status code **and** performing some assertions against the state stored in the database.
Let's split it into two separate test cases:

```
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_returns_a_200_for_valid_form_data() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    // Act
    let response = app.post_subscriptions(body.into()).await;

    // Assert
    assert_eq!(200, response.status().as_u16());
}

#[tokio::test]
async fn subscribe_persists_the_new_subscriber() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    // Act
    app.post_subscriptions(body.into()).await;

    // Assert
    let saved = sqlx::query!("SELECT email, name FROM subscriptions",)
        .fetch_one(&app.db_pool)
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
}
```

We can now modify the second test case to check the status as well.

```
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_persists_the_new_subscriber() {
    // [...]

    // Assert
    let saved = sqlx::query!("SELECT email, name, status FROM subscriptions",)
        .fetch_one(&app.db_pool)
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
    assert_eq!(saved.status, "pending_confirmation");
}
```

The test fails as expected:

```
failures:

---- subscriptions::subscribe_persists_the_new_subscriber stdout ----
thread 'subscriptions::subscribe_persists_the_new_subscriber'
panicked at 'assertion failed: `(left == right)`
  left: `"confirmed"`,
 right: `"pending_confirmation"`'
```

7.7.3.2 Green Test

We can turn it green by touching again our insert query:

```
//! src/routes/subscriptions.rs

#[tracing::instrument([...])]
pub async fn insert_subscriber([...]) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"INSERT INTO subscriptions (id, email, name, subscribed_at, status)
        VALUES ($1, $2, $3, $4, 'confirmed')"#,
        // [...]
    )
    // [...]
}
```

We just need to change `confirmed` into `pending_confirmation`:

```
//! src/routes/subscriptions.rs

#[tracing::instrument([...])]
pub async fn insert_subscriber([...]) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"INSERT INTO subscriptions (id, email, name, subscribed_at, status)
        VALUES ($1, $2, $3, $4, 'pending_confirmation')"#,
        // [...]
    )
  // [...]
}
```

Tests should be green.

7.7.4 Skeleton of `GET /subscriptions/confirm`

We have done most of the groundwork on `POST /subscriptions` - time to shift our focus to the other half of the journey, `GET /subscriptions/confirm`.
We want to build up the skeleton of the endpoint - we need to register the handler against the path in `src/startup.rs` and reject incoming requests without the required query parameter, `subscription_token`.
This will allow us to then build the happy path without having to write a massive amount of code all at once - baby steps!

7.7.4.1 Red Test

Let's add a new module to our `tests` project to host all test cases dealing with the confirmation callback.

```
//! tests/api/main.rs

mod health_check;
mod helpers;
mod subscriptions;
// New module!
mod subscriptions_confirm;
```

```
//! tests/api/subscriptions_confirm.rs
use crate::helpers::spawn_app;

#[tokio::test]
async fn confirmations_without_token_are_rejected_with_a_400() {
    // Arrange
    let app = spawn_app().await;

    // Act
    let response = reqwest::get(&format!("{}/subscriptions/confirm", app.address))
```

```rust
        .await
        .unwrap();

    // Assert
    assert_eq!(response.status().as_u16(), 400);
}
```

Which fails as expected, given that we have no handler yet:

```
---- subscriptions_confirm::confirmations_without_token_are_rejected_with_a_400 stdout ----
thread 'subscriptions_confirm::confirmations_without_token_are_rejected_with_a_400'
panicked at 'assertion failed: `(left == right)`
  left: `404`,
 right: `400`'
```

7.7.4.2 Green Test

Let's start with a dummy handler that returns `200 OK` regardless of the incoming request:

```rust
//! src/routes/mod.rs

mod health_check;
mod subscriptions;
// New module!
mod subscriptions_confirm;

pub use health_check::*;
pub use subscriptions::*;
pub use subscriptions_confirm::*;
```

```rust
//! src/routes/subscriptions_confirm.rs

use actix_web::HttpResponse;

#[tracing::instrument(
    name = "Confirm a pending subscriber",
)]
pub async fn confirm() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

```rust
//! src/startup.rs
// [...]
use crate::routes::confirm;

fn run([...]) -> Result<Server, std::io::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/subscriptions/confirm", web::get().to(confirm))
            // [...]
    })
    // [...]
}
```

We should get a different error now when running `cargo test`:

```
---- subscriptions_confirm::confirmations_without_token_are_rejected_with_a_400 stdout ----
thread 'subscriptions_confirm::confirmations_without_token_are_rejected_with_a_400'
panicked at 'assertion failed: `(left == right)`
  left: `200`,
 right: `400`'
```

It worked!
Time to turn that `200 OK` in a `400 Bad Request`.
We want to ensure that there is a `subscription_token` query parameter: we can rely on another one `actix-web`'s extractors - `Query`.

```rust
//! src/routes/subscriptions_confirm.rs
use actix_web::{HttpResponse, web};

#[derive(serde::Deserialize)]
pub struct Parameters {
    subscription_token: String
}

#[tracing::instrument(
    name = "Confirm a pending subscriber",
    skip(_parameters)
)]
pub async fn confirm(_parameters: web::Query<Parameters>) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

The `Parameters` struct defines all the query parameters that we *expect* to see in the incoming request. It needs to implement `serde::Deserialize` to enable `actix-web` to build it from the incoming request path. It is enough to add a function parameter of type `web::Query<Parameter>` to `confirm` to instruct `actix-web` to only call the handler if the extraction was successful. If the extraction failed a `400 Bad Request` is automatically returned to the caller.

Our test should now pass.

7.7.5 Connecting The Dots

Now that we have a `GET /subscriptions/confirm` handler we can try to perform the full journey!

7.7.5.1 Red Test

We will behave like a user: we will call `POST /subscriptions`, we will extract the confirmation link from the outgoing email request (using the `linkify` machinery we already built) and then call it to confirm our subscription - expecting a `200 OK`.
We will not be checking the `status` from the database (yet) - that is going to be our grand finale.

Let's write it down:

```rust
//! tests/api/subscriptions_confirm.rs
// [...]
use reqwest::Url;
use wiremock::{ResponseTemplate, Mock};
use wiremock::matchers::{path, method};

#[tokio::test]
```

```rust
async fn the_link_returned_by_subscribe_returns_a_200_if_called() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    app.post_subscriptions(body.into()).await;
    let email_request = &app.email_server.received_requests().await.unwrap()[0];
    let body: serde_json::Value = serde_json::from_slice(&email_request.body).unwrap();

    // Extract the link from one of the request fields.
    let get_link = |s: &str| {
        let links: Vec<_> = linkify::LinkFinder::new()
            .links(s)
            .filter(|l| *l.kind() == linkify::LinkKind::Url)
            .collect();
        assert_eq!(links.len(), 1);
        links[0].as_str().to_owned()
    };
    let raw_confirmation_link = &get_link(&body["HtmlBody"].as_str().unwrap());
    let confirmation_link = Url::parse(raw_confirmation_link).unwrap();
    // Let's make sure we don't call random APIs on the web
    assert_eq!(confirmation_link.host_str().unwrap(), "127.0.0.1");

    // Act
    let response = reqwest::get(confirmation_link)
        .await
        .unwrap();

    // Assert
    assert_eq!(response.status().as_u16(), 200);
}
```

It fails with

```
thread 'subscriptions_confirm::the_link_returned_by_subscribe_returns_a_200_if_called'
panicked at 'assertion failed: `(left == right)`
  left: `"my-api.com"`,
 right: `"127.0.0.1"`'
```

There is a fair amount of code duplication going on here, but we will take care of it in due time. Our primary focus is getting the test to pass now.

7.7.5.2 Green Test

Let's begin by taking care of that URL issue.
It is currently hard-coded in

```rust
//! src/routes/subscriptions.rs
// [...]

```

```rust
#[tracing::instrument([...])]
pub async fn send_confirmation_email([...]) -> Result<(), reqwest::Error> {
    let confirmation_link = "https://my-api.com/subscriptions/confirm";
    // [...]
}
```

The domain and the protocol are going to vary according to the environment the application is running into: it will be `http://127.0.0.1` for our tests, it should be a proper DNS record with HTTPS when our application is running in production.
The easiest way to get it right is to pass the domain in as a configuration value.
Let's add a new field to `ApplicationSettings`:

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

#[derive(serde::Deserialize, Clone)]
pub struct ApplicationSettings {
    #[serde(deserialize_with = "deserialize_number_from_string")]
    pub port: u16,
    pub host: String,
    // New field!
    pub base_url: String
}
```

```yaml
# configuration/local.yaml
application:
    base_url: "http://127.0.0.1"
# [...]
```

```yaml
#! spec.yaml
# [...]
services:
  - name: zero2prod
    # [...]
    envs:
      # We use DO's APP_URL to inject the dynamically
      # provisioned base url as an environment variable
      - key: APP_APPLICATION__BASE_URL
        scope: RUN_TIME
        value: ${APP_URL}
        # [...]
# [...]
```

Remember to apply the changes to DigitalOcean every time we touch `spec.yaml`: grab your app identifier via `doctl apps list --format ID` and then run `doctl apps update $APP_ID --spec spec.yaml`.

We now need to register the value in the application context - you should be familiar with the process at this point:

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

impl Application {
    pub async fn build(configuration: Settings) -> Result<Self, std::io::Error> {
        // [...]
        let server = run(
            listener,
            connection_pool,
            email_client,
            // New parameter!
            configuration.application.base_url,
        )?;

        Ok(Self { port, server })
    }

    // [...]
}

// We need to define a wrapper type in order to retrieve the URL
// in the `subscribe` handler.
// Retrieval from the context, in actix-web, is type-based: using
// a raw `String` would expose us to conflicts.
pub struct ApplicationBaseUrl(pub String);

fn run(
    listener: TcpListener,
    db_pool: PgPool,
    email_client: EmailClient,
    // New parameter!
    base_url: String,
) -> Result<Server, std::io::Error> {
    // [...]
    let base_url = Data::new(ApplicationBaseUrl(base_url));
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .app_data(base_url.clone())
    })
    // [...]
}
```

We can now access it in the request handler:

```rust
//! src/routes/subscriptions.rs
use crate::startup::ApplicationBaseUrl;
// [...]

#[tracing::instrument(
    skip(form, pool, email_client, base_url),
    [...]
)]
pub async fn subscribe(
    // [...]
    // New parameter!
    base_url: web::Data<ApplicationBaseUrl>,
) -> HttpResponse {
    // [...]
    // Pass the application url
    if send_confirmation_email(
        &email_client,
        new_subscriber,
        &base_url.0
    )
    .await
    .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    // [...]
}

#[tracing::instrument(
    skip(email_client, new_subscriber, base_url)
    [...]
)]
pub async fn send_confirmation_email(
    // [...]
    // New parameter!
    base_url: &str,
) -> Result<(), reqwest::Error> {
    // Build a confirmation link with a dynamic root
    let confirmation_link = format!("{}/subscriptions/confirm", base_url);
    // [...]
}
```

Let's run the test suite again:

```
thread 'subscriptions_confirm::the_link_returned_by_subscribe_returns_a_200_if_called'
panicked at 'called `Result::unwrap()` on an `Err` value:
    reqwest::Error {
        kind: Request,
        url: Url {
            scheme: "http",
            host: Some(Ipv4(127.0.0.1)),
            port: None,
            path: "/subscriptions/confirm",
            query: None,
            fragment: None },
        source: hyper::Error(
            Connect,
            ConnectError(
                "tcp connect error",
                Os {
                    code: 111,
                    kind: ConnectionRefused,
                    message: "Connection refused"
                }
            )
        )
    }'
```

The host is correct, but the `reqwest::Client` in our test is failing to establish a connection. What is going wrong?

If you look closely, you'll notice `port: None` - we are sending our request to `http://127.0.0.1/subscriptions/confirm` without specifying the port our test server is listening on.

The tricky bit, here, is the sequence of events: we pass in the `application_url` configuration value *before* spinning up the server, therefore we do not know what port it is going to listen to (given that the port is randomised using `0`!).
This is non-issue for production workloads where the DNS domain is enough - we'll just patch it in the test.
Let's store the application port in its own field within `TestApp`:

```rust
//! tests/api/helpers.rs
// [...]

pub struct TestApp {
    // New field!
    pub port: u16,
    // [...]
}

pub async fn spawn_app() -> TestApp {
    // [...]

    let application = Application::build(configuration.clone())
        .await
        .expect("Failed to build application.");
    let application_port = application.port();
    let _ = tokio::spawn(application.run_until_stopped());

    TestApp {
        address: format!("http://localhost:{}", application_port),
        port: application_port,
        db_pool: get_connection_pool(&configuration.database),
        email_server,
    }
}
```

We can then use it in the test logic to edit the confirmation link:

```rust
//! tests/api/subscriptions_confirm.rs
// [...]

#[tokio::test]
async fn the_link_returned_by_subscribe_returns_a_200_if_called() {
    // [...]
    let mut confirmation_link = Url::parse(raw_confirmation_link).unwrap();
    assert_eq!(confirmation_link.host_str().unwrap(), "127.0.0.1");
    // Let's rewrite the URL to include the port
    confirmation_link.set_port(Some(app.port)).unwrap();

    // [...]
}
```

Not the prettiest, but it gets the job done.
Let's run the test again:

```
thread 'subscriptions_confirm::the_link_returned_by_subscribe_returns_a_200_if_called'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `200`'
```

We get a `400 Bad Request` back because our confirmation link does not have a `subscription_token` query parameter attached.
Let's fix it by hard-coding one for the time being:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn send_confirmation_email([...]) -> Result<(), reqwest::Error> {
    let confirmation_link = format!(
        "{}/subscriptions/confirm?subscription_token=mytoken",
        base_url
    );
    // [...]
}
```

The test passes!

7.7.5.3 Refactor

The logic to extract the two confirmation links from the outgoing email request is duplicated across two of our tests - we will likely add more that rely on it as we flesh out the remaining bits and pieces of this feature. It makes sense to extract it in its own helper function.

```rust
//! tests/api/helpers.rs
// [...]

/// Confirmation links embedded in the request to the email API.
pub struct ConfirmationLinks {
    pub html: reqwest::Url,
    pub plain_text: reqwest::Url
}

impl TestApp {
    // [...]

    /// Extract the confirmation links embedded in the request to the email API.
    pub fn get_confirmation_links(
        &self,
        email_request: &wiremock::Request
    ) -> ConfirmationLinks {
        let body: serde_json::Value = serde_json::from_slice(
            &email_request.body
        ).unwrap();

        // Extract the link from one of the request fields.
        let get_link = |s: &str| {
            let links: Vec<_> = linkify::LinkFinder::new()
                .links(s)
                .filter(|l| *l.kind() == linkify::LinkKind::Url)
                .collect();
            assert_eq!(links.len(), 1);
            let raw_link = links[0].as_str().to_owned();
            let mut confirmation_link = reqwest::Url::parse(&raw_link).unwrap();
            // Let's make sure we don't call random APIs on the web
            assert_eq!(confirmation_link.host_str().unwrap(), "127.0.0.1");
            confirmation_link.set_port(Some(self.port)).unwrap();
            confirmation_link
        };

        let html = get_link(&body["HtmlBody"].as_str().unwrap());
        let plain_text = get_link(&body["TextBody"].as_str().unwrap());
        ConfirmationLinks {
            html,
            plain_text
        }
    }
}
```

We are adding it as a method on `TestApp` in order to get access to the application port, which we need to inject into the links.
It could as well have been a free function taking both `wiremock::Request` and `TestApp` (or `u16`) as parameters - a matter of taste.

We can now massively simplify our two test cases:

```rust
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_sends_a_confirmation_email_with_a_link() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    // Act
    app.post_subscriptions(body.into()).await;

    // Assert
    let email_request = &app.email_server.received_requests().await.unwrap()[0];
    let confirmation_links = app.get_confirmation_links(&email_request);

    // The two links should be identical
    assert_eq!(confirmation_links.html, confirmation_links.plain_text);
}
```

```rust
//! tests/api/subscriptions_confirm.rs
// [...]

#[tokio::test]
async fn the_link_returned_by_subscribe_returns_a_200_if_called() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    app.post_subscriptions(body.into()).await;
    let email_request = &app.email_server.received_requests().await.unwrap()[0];
    let confirmation_links = app.get_confirmation_links(&email_request);

    // Act
    let response = reqwest::get(confirmation_links.html)
        .await
        .unwrap();

    // Assert
    assert_eq!(response.status().as_u16(), 200);
}
```

The intent of those two test cases is much clearer now.

7.7.6 Subscription Tokens

We are ready to tackle the elephant in the room: we need to start generating subscription tokens.

7.7.6.1 Red Test

We will add a new test case which builds on top of the work we just did: instead of asserting against the returned status code we will check the `status` of the subscriber stored in the database.

```
//! tests/api/subscriptions_confirm.rs
// [...]

#[tokio::test]
async fn clicking_on_the_confirmation_link_confirms_a_subscriber() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .mount(&app.email_server)
        .await;

    app.post_subscriptions(body.into()).await;
    let email_request = &app.email_server.received_requests().await.unwrap()[0];
    let confirmation_links = app.get_confirmation_links(&email_request);

    // Act
    reqwest::get(confirmation_links.html)
        .await
        .unwrap()
        .error_for_status()
        .unwrap();

    // Assert
    let saved = sqlx::query!("SELECT email, name, status FROM subscriptions",)
        .fetch_one(&app.db_pool)
```

```rust
        .await
        .expect("Failed to fetch saved subscription.");

    assert_eq!(saved.email, "ursula_le_guin@gmail.com");
    assert_eq!(saved.name, "le guin");
    assert_eq!(saved.status, "confirmed");
}
```

The test fails, as expected:

```
thread 'subscriptions_confirm::clicking_on_the_confirmation_link_confirms_a_subscriber'
panicked at 'assertion failed: `(left == right)`
  left: `"pending_confirmation"`,
 right: `"confirmed"`'
```

7.7.6.2 Green Test

To get the previous test case to pass, we hard-coded a subscription token in the confirmation link:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn send_confirmation_email([...]) -> Result<(), reqwest::Error> {
    let confirmation_link = format!(
        "{}/subscriptions/confirm?subscription_token=mytoken",
        base_url
    );
    // [...]
}
```

Let's refactor `send_confirmation_email` to take the token as a parameter - it will make it easier to add the generation logic upstream.

```rust
//! src/routes/subscriptions.rs
// [...]

#[tracing::instrument([...])]
pub async fn subscribe([...]) -> HttpResponse {
    // [...]
    if send_confirmation_email(
        &email_client,
        new_subscriber,
        &base_url.0,
        // New parameter!
        "mytoken"
    )
    .await
    .is_err() {
        return HttpResponse::InternalServerError().finish();
    }
    // [...]
}

#[tracing::instrument(
    name = "Send a confirmation email to a new subscriber",
    skip(email_client, new_subscriber, base_url, subscription_token)
)]
```

```rust
pub async fn send_confirmation_email(
    email_client: &EmailClient,
    new_subscriber: NewSubscriber,
    base_url: &str,
    // New parameter!
    subscription_token: &str
) -> Result<(), reqwest::Error> {
    let confirmation_link = format!(
        "{}/subscriptions/confirm?subscription_token={}",
        base_url,
        subscription_token
    );
    // [...]
}
```

Our subscription tokens are not passwords: they are single-use and they do not grant access to protected information.[6] We need them to be hard enough to guess while keeping in mind that the worst-case scenario is an unwanted newsletter subscription landing in someone's inbox.

Given our requirements it should be enough to use a cryptographically secure pseudo-random number generator - a *CSPRNG*, if you are into obscure acronyms.
Every time we need to generate a subscription token we can sample a sufficiently-long sequence of alphanumeric characters.

To pull it off we need to add `rand` as a dependency:

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

[dependencies]
# [...]
# We need the `std_rng` to get access to the PRNG we want
rand = { version = "0.8", features=["std_rng"] }
```

```rust
//! src/routes/subscriptions.rs
use rand::distributions::Alphanumeric;
use rand::{thread_rng, Rng};
// [...]

/// Generate a random 25-characters-long case-sensitive subscription token.
fn generate_subscription_token() -> String {
  let mut rng = thread_rng();
  std::iter::repeat_with(|| rng.sample(Alphanumeric))
          .map(char::from)
          .take(25)
          .collect()
}
```

Using 25 characters we get roughly ~10^45 possible tokens - it should be more than enough for our use case.
To check if a token is valid in `GET /subscriptions/confirm` we need `POST /subscriptions` to store the newly minted tokens in the database.

[6] You could say that our token is a nonce.

The table we added for this purpose, `subscription_tokens`, has two columns: `subscription_token` and `subscriber_id`.

We are currently generating the subscriber identifier in `insert_subscriber` but we never return it to the caller:

```rust
#[tracing::instrument([...])]
pub async fn insert_subscriber([...]) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"[...]"#,
        // The subscriber id, never returned or bound to a variable
        Uuid::new_v4(),
        // [...]
    )
    // [...]
}
```

Let's refactor `insert_subscriber` to give us back the identifier:

```rust
#[tracing::instrument([...])]
pub async fn insert_subscriber([...]) -> Result<Uuid, sqlx::Error> {
    let subscriber_id = Uuid::new_v4();
    sqlx::query!(
        r#"[...]"#,
        subscriber_id,
        // [...]
    )
    // [...]
    Ok(subscriber_id)
}
```

We can now tie everything together:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe([...]) -> HttpResponse {
    // [...]
    let subscriber_id = match insert_subscriber(&pool, &new_subscriber).await {
        Ok(subscriber_id) => subscriber_id,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    let subscription_token = generate_subscription_token();
    if store_token(&pool, subscriber_id, &subscription_token)
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    if send_confirmation_email(
        &email_client,
        new_subscriber,
        &base_url.0,
        &subscription_token,
    )
    .await
    .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    HttpResponse::Ok().finish()
}


#[tracing::instrument(
    name = "Store subscription token in the database",
    skip(subscription_token, pool)
)]
pub async fn store_token(
    pool: &PgPool,
    subscriber_id: Uuid,
    subscription_token: &str,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"INSERT INTO subscription_tokens (subscription_token, subscriber_id)
        VALUES ($1, $2)"#,
        subscription_token,
        subscriber_id
    )
    .execute(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

We are done on `POST /subscriptions`, let's shift to `GET /subscription/confirm`:

```
//! src/routes/subscriptions_confirm.rs
use actix_web::{HttpResponse, web};

#[derive(serde::Deserialize)]
pub struct Parameters {
    subscription_token: String
}

#[tracing::instrument(
    name = "Confirm a pending subscriber",
    skip(_parameters)
)]
pub async fn confirm(_parameters: web::Query<Parameters>) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

We need to:

- get a reference to the database pool;
- retrieve the subscriber id associated with the token (if one exists);
- change the subscriber `status` to `confirmed`.

Nothing we haven't done before - let's get cracking!

```rust
use actix_web::{web, HttpResponse};
use sqlx::PgPool;
use uuid::Uuid;

#[derive(serde::Deserialize)]
pub struct Parameters {
    subscription_token: String,
}

#[tracing::instrument(
    name = "Confirm a pending subscriber",
    skip(parameters, pool)
)]
pub async fn confirm(
    parameters: web::Query<Parameters>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let id = match get_subscriber_id_from_token(&pool, &parameters.subscription_token).await {
        Ok(id) => id,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    match id {
        // Non-existing token!
        None => HttpResponse::Unauthorized().finish(),
        Some(subscriber_id) => {
            if confirm_subscriber(&pool, subscriber_id).await.is_err() {
                return HttpResponse::InternalServerError().finish();
            }
            HttpResponse::Ok().finish()
        }
    }
}

#[tracing::instrument(
    name = "Mark subscriber as confirmed",
    skip(subscriber_id, pool)
)]
pub async fn confirm_subscriber(
    pool: &PgPool,
    subscriber_id: Uuid
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"UPDATE subscriptions SET status = 'confirmed' WHERE id = $1"#,
        subscriber_id,
        )
        .execute(pool)
        .await
        .map_err(|e| {
            tracing::error!("Failed to execute query: {:?}", e);
            e
        })?;
    Ok(())
}

#[tracing::instrument(
    name = "Get subscriber_id from token",
    skip(subscription_token, pool)
)]
pub async fn get_subscriber_id_from_token(
    pool: &PgPool,
    subscription_token: &str,
) -> Result<Option<Uuid>, sqlx::Error> {
    let result = sqlx::query!(
      r#"SELECT subscriber_id FROM subscription_tokens WHERE subscription_token = $1"#,
      subscription_token,
    )
    .fetch_optional(pool)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(result.map(|r| r.subscriber_id))
}
```

Is it enough? Did we miss anything during the journey?
There is only one way to find out.

```
cargo test
```

```
     Running target/debug/deps/api-5a717281b98f7c41
running 10 tests
[...]

test result: ok. 10 passed; 0 failed; finished in 0.92s
```

Oh, yes! It works!

7.8 Database Transactions

7.8.1 All Or Nothing

It is too soon to declare victory though.
Our `POST /subscriptions` handler has grown in complexity - we are now performing two `INSERT` queries against our Postgres database: one to store the details of the new subscriber, one to store the newly generated subscription token.
What happens if the application crashes between those two operations?

The first query might complete successfully, but the second one might never be executed.

There are three possible states for our database after an invocation of `POST /subscriptions`:

- a new subscriber and its token have been persisted;
- a new subscriber has been persisted, without a token;
- nothing has been persisted.

The more queries you have, the worse it gets to reason about the possible end states of our database.

Relational databases (and a few others) provide a mechanism to mitigate this issue: **transactions**.

Transactions are a way to group together related operations in a single **unit of work**.

The database guarantees that all operations within a transaction will succeed or fail together: the database will never be left in a state where the effect of only a subset of the queries in a transaction is visible.

Going back to our example, if we wrap the two `INSERT` queries in a transaction we now have **two** possible end states:

- a new subscriber and its token have been persisted;
- nothing has been persisted.

Much easier to deal with.

7.8.2 Transactions In Postgres

To start a transaction in Postgres you use a `BEGIN` statement. All queries after `BEGIN` are part of the transaction.
The transaction is then finalised with a `COMMIT` statement.

We have actually already used a transaction in one of our migration scripts!

```sql
BEGIN;
UPDATE subscriptions SET status = 'confirmed' WHERE status IS NULL;
ALTER TABLE subscriptions ALTER COLUMN status SET NOT NULL;
COMMIT;
```

If any of the queries within a transaction fails the database **rolls back**: all changes performed by previous queries are reverted, the operation is aborted.
You can also explicitly trigger a rollback with the `ROLLBACK` statement.

Transactions are a deep topic: they not only provide a way to convert multiple statements into an all-or-nothing operation, they also hide the effect of uncommitted changes from other queries that might be running, concurrently, against the same tables.
As your needs evolves, you will often want to explicitly choose the **isolation level** of your transactions to fine-tune the concurrency guarantees provided by the database on your operations. Getting a good grip on the different kinds of concurrency-related issues (e.g. dirty reads, phantom reads, etc.) becomes more and more important as your system grows in scale and complexity.

I can't recommend "Designing Data Intensive Applications" enough if you want to learn more about these topics.

7.8.3 Transactions In Sqlx

Back to the code: how do we leverage transactions in `sqlx`?

You don't have to manually write a `BEGIN` statement: transactions are so central to the usage of relational databases that `sqlx` provides a dedicated API.
By calling `begin` on our `pool` we acquire a connection from the pool and kick off a transaction:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe([...]) -> HttpResponse {
    let new_subscriber = // [...]
    let mut transaction = match pool.begin().await {
        Ok(transaction) => transaction,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    // [...]
```

`begin`, if successful, returns a `Transaction` struct.
A mutable reference to a `Transaction` implements `sqlx`'s `Executor` trait therefore it can be used to run queries. All queries run using a `Transaction` as executor become of the transaction.

Let's pass `transaction` down to `insert_subscriber` and `store_token` instead of `pool`:

```rust
//! src/routes/subscriptions.rs
use sqlx::{Postgres, Transaction};
// [...]

#[tracing::instrument([...])]
pub async fn subscribe([...]) -> HttpResponse {
    // [...]
    let mut transaction = match pool.begin().await {
        Ok(transaction) => transaction,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    let subscriber_id = match insert_subscriber(&mut transaction, &new_subscriber).await {
        Ok(subscriber_id) => subscriber_id,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    let subscription_token = generate_subscription_token();
    if store_token(&mut transaction, subscriber_id, &subscription_token)
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    // [...]
}

#[tracing::instrument(
    name = "Saving new subscriber details in the database",
    skip(new_subscriber, transaction)
)]
pub async fn insert_subscriber(
    transaction: &mut Transaction<'_, Postgres>,
    new_subscriber: &NewSubscriber,
) -> Result<Uuid, sqlx::Error> {
    let subscriber_id = Uuid::new_v4();
    sqlx::query!([...])
        .execute(transaction)
    // [...]
}

#[tracing::instrument(
    name = "Store subscription token in the database",
    skip(subscription_token, transaction)
)]
pub async fn store_token(
    transaction: &mut Transaction<'_, Postgres>,
    subscriber_id: Uuid,
    subscription_token: &str,
) -> Result<(), sqlx::Error> {
    sqlx::query!([..])
        .execute(transaction)
    // [...]
}
```

If you run `cargo test` now you will see something funny: some of our tests are failing! Why is that happening?

As we discussed, a transaction has to either be committed or rolled back.
`Transaction` exposes two dedicated methods: `Transaction::commit`, to persist changes, and `Transaction::rollback`, to abort the whole operation.
We are not calling either - what happens in that case?

We can look at `sqlx`'s source code to understand better.
In particular, `Transaction`'s `Drop` implementation:

```
impl<'c, DB> Drop for Transaction<'c, DB>
where
    DB: Database,
{
    fn drop(&mut self) {
        if self.open {
            // starts a rollback operation

            // what this does depends on the database but generally
            // this means we queue a rollback operation that will
            // happen on the next asynchronous invocation of the
            // underlying connection (including if the connection
            // is returned to a pool)
            DB::TransactionManager::start_rollback(&mut self.connection);
        }
    }
}
```

`self.open` is an internal boolean flag attached to the connection used to begin the transaction and run the queries attached to it.
When a transaction is created, using `begin`, it is set to `true` until either `rollback` or `commit` are called:

```
impl<'c, DB> Transaction<'c, DB>
where
    DB: Database,
{
    pub(crate) fn begin(
        conn: impl Into<MaybePoolConnection<'c, DB>>,
    ) -> BoxFuture<'c, Result<Self, Error>> {
        let mut conn = conn.into();

        Box::pin(async move {
            DB::TransactionManager::begin(&mut conn).await?;

            Ok(Self {
                connection: conn,
                open: true,
```

```rust
            })
        })
    }

    pub async fn commit(mut self) -> Result<(), Error> {
        DB::TransactionManager::commit(&mut self.connection).await?;
        self.open = false;

        Ok(())
    }

    pub async fn rollback(mut self) -> Result<(), Error> {
        DB::TransactionManager::rollback(&mut self.connection).await?;
        self.open = false;

        Ok(())
    }
}
```

In other words: if `commit` or `rollback` have not been called before the `Transaction` object goes out of scope (i.e. `Drop` is invoked), a `rollback` command is queued to be executed as soon as an opportunity arises.[7]
That is why our tests are failing: we are using a transaction but we are not explicitly committing the changes. When the connection goes back into the pool, at the end of our request handler, all changes are rolled back and our test expectations are not met.

We can fix it by adding a one-liner to `subscribe`:

```rust
//! src/routes/subscriptions.rs
use sqlx::{Postgres, Transaction};
// [...]

#[tracing::instrument([...])]
pub async fn subscribe([...]) -> HttpResponse {
    // [...]
    let mut transaction = match pool.begin().await {
        Ok(transaction) => transaction,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    let subscriber_id = match insert_subscriber(&mut transaction, &new_subscriber).await {
        Ok(subscriber_id) => subscriber_id,
        Err(_) => return HttpResponse::InternalServerError().finish(),
    };
    let subscription_token = generate_subscription_token();
    if store_token(&mut transaction, subscriber_id, &subscription_token)
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
```

[7] Rust does not currently support asynchronous destructors, a.k.a. `AsyncDrop`. There have been some discussions on the topic, but there is no consensus yet. This is a constraint on `sqlx`: when `Transaction` goes out of scope it can enqueue a rollback operation, but it cannot execute it immediately! Is it ok? Is it a sound API? There are different views - see `diesel`'s async issue for an overview. My personal view is that the benefits brought by `sqlx` to the table offset the risks, but you should make an informed decision taking into account the tradeoffs of your application and use case.

```
    }
    if transaction.commit().await.is_err() {
        return HttpResponse::InternalServerError().finish();
    }
    // [...]
}
```

The test suite should succeed once again.
Go ahead and deploy the application: seeing a feature working in a live environment adds a whole new level of satisfaction!

7.9 Summary

This chapter was a long journey, but you have come a long way as well!
The skeleton of our application has started to shape up, starting with our test suite. Features are moving along as well: we now have a functional subscription flow, with a proper confirmation email.
More importantly: we are getting into the **rhythm** of writing Rust code.
The very end of the chapter has been a long pair programming session where we have made significant progress *without* introducing many new concepts.

This is a great moment to go off and explore a bit on your own: improve on what we built so far!
There are plenty of opportunities:

- What happens if a user tries to subscribe twice? Make sure that they receive two confirmation emails;
- What happens if a user clicks on a confirmation link twice?
- What happens if the subscription token is well-formatted but non-existent?
- Add validation on the incoming token, we are currently passing the raw user input straight into a query (thanks `sqlx` for protecting us from SQL injections <3);
- Use a proper templating solution for our emails (e.g. `tera`);
- Anything that comes to your mind!

It takes deliberate practice to achieve mastery.

Chapter 8

Error Handling

To send a confirmation email we had to stitch together multiple operations: validation of user input, email dispatch, various database queries.
They all have one thing in common: they may fail.

In Chapter 6 we discussed the building blocks of error handling in Rust - `Result` and the `?` operator.
We left many questions unanswered: how do errors fit within the broader architecture of our application? What does a *good error* look like? Who are errors for? Should we use a library? Which one?

An in-depth analysis of error handling patterns in Rust will be the sole focus of this chapter.

8.1 What Is The Purpose Of Errors?

Let's start with an example:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn store_token(
    transaction: &mut Transaction<'_, Postgres>,
    subscriber_id: Uuid,
    subscription_token: &str,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
    INSERT INTO subscription_tokens (subscription_token, subscriber_id)
    VALUES ($1, $2)
        "#,
        subscription_token,
        subscriber_id
    )
    .execute(transaction)
    .await
    .map_err(|e| {
        tracing::error!("Failed to execute query: {:?}", e);
        e
    })?;
    Ok(())
}
```

We are trying to insert a row into the `subscription_tokens` table in order to store a newly-generated token against a `subscriber_id`.
`execute` is a fallible operation: we might have a network issue while talking to the database, the row we are trying to insert might violate some table constraints (e.g. uniqueness of the primary key), etc.

8.1.1 Internal Errors

8.1.1.1 Enable The Caller To React

The caller of `execute` most likely wants to be informed if a failure occurs - **they need to *react* accordingly**, e.g. retry the query or propagate the failure upstream using `?`, as in our example.

Rust leverages the type system to communicate that an operation may not succeed: the return type of `execute` is `Result`, an enum.

```
pub enum Result<Success, Error> {
    Ok(Success),
    Err(Error)
}
```

The caller is then forced by the compiler to express how they plan to handle both scenarios - success and failure.

If our only goal was to communicate to the caller that an error happened, we could use a simpler definition for `Result`:

```
pub enum ResultSignal<Success> {
    Ok(Success),
    Err
}
```

There would be no need for a generic `Error` type - we could just check that `execute` returned the `Err` variant, e.g.

```
let outcome = sqlx::query!(/* ... */)
    .execute(transaction)
    .await;
if outcome == ResultSignal::Err {
    // Do something if it failed
}
```

This works if there is only one failure mode. Truth is, operations can fail in *multiple* ways and we might want to react *differently* depending on what happened.
Let's look at the skeleton of `sqlx::Error`, the error type for `execute`:

```
//! sqlx-core/src/error.rs

pub enum Error {
    Configuration(/* */),
    Database(/* */),
    Io(/* */),
    Tls(/* */),
```

```rust
    Protocol(/* */),
    RowNotFound,
    TypeNotFound {/* */},
    ColumnIndexOutOfBounds {/* */},
    ColumnNotFound(/* */),
    ColumnDecode {/* */},
    Decode(/* */),
    PoolTimedOut,
    PoolClosed,
    WorkerCrashed,
    Migrate(/* */),
}
```

Quite a list, ain't it?
`sqlx::Error` is implemented as an enum to allow users to match on the returned error and behave *differently* depending on the underlying failure mode. For example, you might want to retry a `PoolTimedOut` while you will probably give up on a `ColumnNotFound`.

8.1.1.2 Help An Operator To Troubleshoot

What if an operation has a single failure mode - should we just use `()` as error type?

`Err(())` might be enough for the caller to determine what to do - e.g. return a `500 Internal Server Error` to the user.

But control flow is not the *only* purpose of errors in an application.
We expect errors to carry enough **context** about the failure to produce a **report** for an operator (e.g. the developer) that contains enough details to go and troubleshoot the issue.

What do we mean by report?
In a backend API like ours it will usually be a log event.
In a CLI it could be an error message shown in the terminal when a `--verbose` flag is used.

The implementation details may vary, the purpose stays the same: help a **human** understand what is going wrong.
That's exactly what we are doing in the initial code snippet:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn store_token(/* */) -> Result<(), sqlx::Error> {
    sqlx::query!(/* */)
        .execute(transaction)
        .await
        .map_err(|e| {
            tracing::error!("Failed to execute query: {:?}", e);
            e
        })?;
    // [...]
}
```

If the query fails, we grab the error and emit a log event. We can then go and inspect the error logs when investigating the database issue.

8.1.2 Errors At The Edge

8.1.2.1 Help A User To Troubleshoot

So far we focused on the internals of our API - functions calling other functions and operators trying to make sense of the mess after it happened.
What about users?

Just like operators, users expect the API to **signal** when a failure mode is encountered.

What does a user of our API see when `store_token` fails?
We can find out by looking at the request handler:

```
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    // [...]
    if store_token(&mut transaction, subscriber_id, &subscription_token)
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    // [...]
}
```

They receive an HTTP response with no body and a `500 Internal Server Error` status code.

The status code fulfills the same purpose of the error type in `store_token`: it is a machine-parsable piece of information that the caller (e.g. the browser) can use to determine what to do next (e.g. retry the request assuming it's a transient failure).

What about the human behind the browser? What are we telling them?
Not much, the response body is empty.
That is actually a good implementation: the user should not have to care about the internals of the API they are calling - they have no mental model of it and no way to determine why it is failing. That's the realm of the operator.
We are **omitting** those details by design.

In other circumstances, instead, we *need* to convey additional information to the human user. Let's look at our input validation for the same endpoint:

```
//! src/routes/subscriptions.rs

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

impl TryFrom<FormData> for NewSubscriber {
    type Error = String;

    fn try_from(value: FormData) -> Result<Self, Self::Error> {
        let name = SubscriberName::parse(value.name)?;
        let email = SubscriberEmail::parse(value.email)?;
```

```rust
        Ok(Self { email, name })
    }
}
```

We received an email address and a name as data attached to the form submitted by the user. Both fields are going through an additional round of validation - `SubscriberName::parse` and `SubscriberEmail::parse`. Those two methods are fallible - they return a `String` as error type to explain what has gone wrong:

```rust
//! src/domain/subscriber_email.rs
// [...]

impl SubscriberEmail {
    pub fn parse(s: String) -> Result<SubscriberEmail, String> {
        if validate_email(&s) {
            Ok(Self(s))
        } else {
            Err(format!("{} is not a valid subscriber email.", s))
        }
    }
}
```

It is, I must admit, not the most useful error message: we are telling the user that the email address they entered is wrong, but we are not helping them to determine *why*.
In the end, it doesn't matter: we are not sending any of that information to the user as part of the response of the API - they are getting a `400 Bad Request` with no body.

```rust
//! src/routes/subscription.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    let new_subscriber = match form.0.try_into() {
        Ok(form) => form,
        Err(_) => return HttpResponse::BadRequest().finish(),
    };
    // [...]
```

This is a poor error: the user is left in the dark and cannot adapt their behaviour as required.

8.1.3 Summary

Let's summarise what we uncovered so far.
Errors serve two[1] main purposes:

- Control flow (i.e. determine what do next);
- Reporting (e.g. investigate, *after the fact*, what went wrong on).

We can also distinguish errors based on their location:

- Internal (i.e. a function calling another function within our application);
- At the edge (i.e. an API request that we failed to fulfill).

[1] We are borrowing the terminology introduced by Jane Lusby in "Error handling Isn't All About Errors", a talk from RustConf 2020. If you haven't watched it yet, close the book and open YouTube - you will not regret it.

Control flow is scripted: all information required to take a decision on what to do next must be accessible to a **machine**.
We use types (e.g. enum variants), methods and fields for internal errors.
We rely on status codes for errors at the edge.

Error reports, instead, are primarily consumed by **humans**.
The content has to be tuned depending on the audience.
An operator has access to the internals of the system - they should be provided with as much **context** as possible on the failure mode.
A user sits outside the boundary of the application[2]: they should only be given the amount of information required to adjust *their* behaviour if necessary (e.g. fix malformed inputs).

We can visualise this mental model using a 2x2 table with `Location` as columns and `Purpose` as rows:

	Internal	At the edge
Control Flow	Types, methods, fields	Status codes
Reporting	Logs/traces	Response body

We will spend the rest of the chapter improving our error handling strategy for each of the cells in the table.

8.2 Error Reporting For Operators

Let's start with error reporting for operators.
Are we doing a good job with logging right now when it comes to errors?

Let's write a quick test to find out:

```rust
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_fails_if_there_is_a_fatal_database_error() {
    // Arrange
    let app = spawn_app().await;
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";
    // Sabotage the database
    sqlx::query!("ALTER TABLE subscription_tokens DROP COLUMN subscription_token;",)
        .execute(&app.db_pool)
        .await
        .unwrap();

    // Act
    let response =  app.post_subscriptions(body.into()).await;
```

[2] It is good to keep in mind that the line between a user and an operator can be blurry - e.g. a user might have access to the source code or they might be running the software on their own hardware. They might have to wear the operator's hat at times. For similar scenarios there should be configuration knobs (e.g. `--verbose` or an environment variable for a CLI) to clearly inform the software of the human **intent** so that it can provide diagnostics at the right level of detail and abstraction.

```
    // Assert
    assert_eq!(response.status().as_u16(), 500);
}
```

The test passes straight away - let's look at the log emitted by the application[3].

> Make sure you are running on `tracing-actix-web 0.4.0-beta.8`, `tracing-bunyan-formatter 0.2.4` and `actix-web 4.0.0-beta.8`!

```
# sqlx logs are a bit spammy, cutting them out to reduce noise
export RUST_LOG="sqlx=error,info"
export TEST_LOG=enabled
cargo t subscribe_fails_if_there_is_a_fatal_database_error | bunyan
```

The output, once you focus on what matters, is the following:

```
 INFO: [HTTP REQUEST - START]
 INFO: [ADDING A NEW SUBSCRIBER - START]
 INFO: [SAVING NEW SUBSCRIBER DETAILS IN THE DATABASE - START]
 INFO: [SAVING NEW SUBSCRIBER DETAILS IN THE DATABASE - END]
 INFO: [STORE SUBSCRIPTION TOKEN IN THE DATABASE - START]
ERROR: [STORE SUBSCRIPTION TOKEN IN THE DATABASE - EVENT] Failed to execute query:
        Database(PgDatabaseError {
          severity: Error,
          code: "42703",
          message:
            "column 'subscription_token' of relation
             'subscription_tokens' does not exist",
          ...
        })
    target=zero2prod::routes::subscriptions
 INFO: [STORE SUBSCRIPTION TOKEN IN THE DATABASE - END]
 INFO: [ADDING A NEW SUBSCRIBER - END]
ERROR: [HTTP REQUEST - EVENT] Error encountered while
        processing the incoming HTTP request: ""
    exception.details="",
    exception.message="",
    target=tracing_actix_web::middleware
 INFO: [HTTP REQUEST - END]
    exception.details="",
    exception.message="",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

How do you read something like this?
Ideally, you start from the outcome: the log record emitted at the end of request processing. In

[3] In an ideal scenario we would actually be writing a test to verify the properties of the logs emitted by our application. This is somewhat cumbersome to do today - I am looking forward to revising this chapter when better tooling becomes available (or I get nerd-sniped into writing it).

our case, that is:

```
 INFO: [HTTP REQUEST - END]
    exception.details="",
    exception.message="",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

What does that tell us?
The request returned a `500` status code - it failed.
We don't learn a lot more than that: both `exception.details` and `exception.message` are empty.

The situation does not get much better if we look at the next log, emitted by `tracing_actix_web`:

```
ERROR: [HTTP REQUEST - EVENT] Error encountered while
        processing the incoming HTTP request: ""
    exception.details="",
    exception.message="",
    target=tracing_actix_web::middleware
```

No actionable information whatsoever. Logging "Oops! Something went wrong!" would have been just as useful.

We need to keep looking, all the way to the last remaining error log:

```
ERROR: [STORE SUBSCRIPTION TOKEN IN THE DATABASE - EVENT] Failed to execute query:
        Database(PgDatabaseError {
          severity: Error,
          code: "42703",
          message:
            "column 'subscription_token' of relation
             'subscription_tokens' does not exist",
          ...
        })
    target=zero2prod::routes::subscriptions
```

Something went wrong when we tried talking to the database - we were expecting to see a `subscription_token` column in the `subscription_tokens` table but, for some reason, it was not there.
This is actually useful!

Is it the cause of the `500` though?
Difficult to say just by looking at the logs - a developer will have to clone the codebase, check where that log line is coming from and make sure that it's indeed the cause of the issue.
It can be done, but it takes time: it would be much easier if the `[HTTP REQUEST - END]` log record reported something useful about the **underlying root cause** in `exception.details` and `exception.message`.

8.2.1 Keeping Track Of The Error Root Cause

To understand why the log records coming out `tracing_actix_web` are so poor we need to inspect (again) our request handler and `store_token`:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> HttpResponse {
    // [...]
    if store_token(&mut transaction, subscriber_id, &subscription_token)
        .await
        .is_err()
    {
        return HttpResponse::InternalServerError().finish();
    }
    // [...]
}

pub async fn store_token(/* */) -> Result<(), sqlx::Error> {
    sqlx::query!(/* */)
        .execute(transaction)
        .await
        .map_err(|e| {
            tracing::error!("Failed to execute query: {:?}", e);
            e
        })?;
    // [...]
}
```

The useful error log we found is indeed the one emitted by that `tracing::error` call - the error message includes the `sqlx::Error` returned by `execute`.
We propagate the error upwards using the `?` operator, but the chain breaks in `subscribe` - we discard the error we received from `store_token` and build a bare `500` response.

`HttpResponse::InternalServerError().finish()` is the only thing that `actix_web` and `tracing_actix_web::TracingLogger` get to access when they are about to emit their respective log records. The error does not contain any context about the **underlying root cause**, therefore the log records are equally useless.

How do we fix it?

We need to start leveraging the error handling machinery exposed by `actix_web` - in particular, `actix_web::Error`. According to the documentation:

> `actix_web::Error` is used to carry errors from `std::error` through `actix_web` in a convenient way.

It sounds exactly like what we are looking for. How do we build an instance of `actix_web::Error`?
The documentation states that

> `actix_web::Error` can be created by converting errors with `into()`.

A bit indirect, but we can figure it out[4].
The only `From`/`Into` implementation that we can use, browsing the ones listed in the documentation, seems to be this one:

[4] I pinky-swear that I am going to submit a PR to `actix_web` to improve this section of the documentation.

```rust
/// Build an `actix_web::Error` from any error that implements `ResponseError`
impl<T: ResponseError + 'static> From<T> for Error {
    fn from(err: T) -> Error {
        Error {
            cause: Box::new(err),
        }
    }
}
```

ResponseError is a trait exposed by `actix_web`:

```rust
/// Errors that can be converted to `Response`.
pub trait ResponseError: fmt::Debug + fmt::Display {
    /// Response's status code.
    ///
    /// The default implementation returns an internal server error.
    fn status_code(&self) -> StatusCode;

    /// Create a response from the error.
    ///
    /// The default implementation returns an internal server error.
    fn error_response(&self) -> Response;
}
```

We just need to implement it for our errors!
`actix_web` provides a default implementation for both methods that returns a `500 Internal Server Error` - exactly what we need. Therefore it's enough to write:

```rust
//! src/routes/subscriptions.rs
use actix_web::ResponseError;
// [...]

impl ResponseError for sqlx::Error {}
```

The compiler is not happy:

```
error[E0117]: only traits defined in the current crate
              can be implemented for arbitrary types
   --> src/routes/subscriptions.rs:162:1
    |
162 | impl ResponseError for sqlx::Error {}
    | ^^^^^^^^^^^^^^^^^^^^^^^-----------
    | |                      |
    | |                      `sqlx::Error` is not defined in the current crate
    | impl doesn't use only types from inside the current crate
    |
    = note: define and implement a trait or new type instead
```

We just bumped into Rust's orphan rule: it is forbidden to implement a foreign trait for a foreign type, where foreign stands for "from another crate".
This restriction is meant to preserve coherence: imagine if you added a dependency that defined its own implementation of `ResponseError` for `sqlx::Error` - which one should the compiler use when the trait methods are invoked?

Orphan rule aside, it would still be a mistake for us to implement `ResponseError` for `sqlx::Error`.
We want to return a `500 Internal Server Error` when we run into a `sqlx::Error` *while trying*

to persist a subscriber token.
In another circumstance we might wish to handle a `sqlx::Error` differently.

We should follow the compiler's suggestion: define a new type to wrap `sqlx::Error`.

```rust
//! src/routes/subscriptions.rs
// [...]

//                                    Using the new error type!
pub async fn store_token(/* */) -> Result<(), StoreTokenError> {
    sqlx::query!(/* */)
    .execute(transaction)
    .await
    .map_err(|e| {
        // [...]
        // Wrapping the underlying error
        StoreTokenError(e)
    })?;
    // [...]
}

// A new error type, wrapping a sqlx::Error
pub struct StoreTokenError(sqlx::Error);

impl ResponseError for StoreTokenError {}
```

It doesn't work, but for a different reason:

```
error[E0277]: `StoreTokenError` doesn't implement `std::fmt::Display`
   --> src/routes/subscriptions.rs:164:6
    |
164 | impl ResponseError for StoreTokenError {}
    |      ^^^^^^^^^^^^^
    `StoreTokenError` cannot be formatted with the default formatter
    |
    |
59  | pub trait ResponseError: fmt::Debug + fmt::Display {
    |                                       ------------
    |          required by this bound in `ResponseError`
    |
    = help: the trait `std::fmt::Display` is not implemented for `StoreTokenError`

error[E0277]: `StoreTokenError` doesn't implement `std::fmt::Debug`
   --> src/routes/subscriptions.rs:164:6
    |
164 | impl ResponseError for StoreTokenError {}
    |      ^^^^^^^^^^^^^
    `StoreTokenError` cannot be formatted using `{:?}`
    |
    |
59  | pub trait ResponseError: fmt::Debug + fmt::Display {
    |                          ----------
             required by this bound in `ResponseError`
    |
    = help: the trait `std::fmt::Debug` is not implemented for `StoreTokenError`
    = note: add `#[derive(Debug)]` or manually implement `std::fmt::Debug`
```

We are missing two trait implementations on `StoreTokenError`: `Debug` and `Display`.

Both traits are concerned with formatting, but they serve a different purpose.
`Debug` should return a programmer-facing representation, as faithful as possible to the underlying type structure, to help with debugging (as the name implies). Almost all public types should implement `Debug`.
`Display`, instead, should return a user-facing representation of the underlying type. Most types do not implement `Display` and it cannot be automatically implemented with a `#[derive(Display)]` attribute.

When working with errors, we can reason about the two traits as follows: `Debug` returns as much information as possible while `Display` gives us a brief description of the failure we encountered, with the essential amount of context.

Let's give it a go for `StoreTokenError`:

```rust
//! src/routes/subscriptions.rs
// [...]

// We derive `Debug`, easy and painless.
#[derive(Debug)]
pub struct StoreTokenError(sqlx::Error);

impl std::fmt::Display for StoreTokenError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "A database error was encountered while \
            trying to store a subscription token."
        )
    }
}
```

It compiles!
We can now leverage it in our request handler:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> Result<HttpResponse, actix_web::Error> {
    // You will have to wrap (early) returns in `Ok(...)` as well!
    // [...]
    // The `?` operator transparently invokes the `Into` trait
    // on our behalf - we don't need an explicit `map_err` anymore.
    store_token(/* */).await?;
    // [...]
}
```

Let's look at our logs again:

```bash
# sqlx logs are a bit spammy, cutting them out to reduce noise
export RUST_LOG="sqlx=error,info"
export TEST_LOG=enabled
cargo t subscribe_fails_if_there_is_a_fatal_database_error | bunyan
```

```
...
 INFO: [HTTP REQUEST - END]
    exception.details= StoreTokenError(
        Database(
            PgDatabaseError {
                severity: Error,
                code: "42703",
                message:
                    "column 'subscription_token' of relation
                     'subscription_tokens' does not exist",
                ...
            }
        )
    )
    exception.message=
        "A database failure was encountered while
         trying to store a subscription token.",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

Much better!
The log record emitted at the end of request processing now contains both an in-depth and brief description of the error that caused the application to return a `500 Internal Server Error` to the user.
It is enough to look at this log record to get a pretty accurate picture of everything that matters for this request.

8.2.2 The `Error` Trait

So far we moved forward by following the compiler suggestions, trying to satisfy the constraints imposed on us by `actix-web` when it comes to error handling.
Let's step back to look at the bigger picture: what should an error look like in Rust (not considering the specifics of `actix-web`)?

Rust's standard library has a dedicated trait, `Error`.

```rust
pub trait Error: Debug + Display {
    /// The lower-level source of this error, if any.
    fn source(&self) -> Option<&(dyn Error + 'static)> {
        None
    }
}
```

It requires an implementation of `Debug` and `Display`, just like `ResponseError`.
It also gives us the option to implement a `source` method that returns the underlying cause of the error, if any.

What is the point of implementing the `Error` trait at all for our error type?
It is not required by `Result` - any type can be used as error variant there.

```rust
pub enum Result<T, E> {
    /// Contains the success value
    Ok(T),

    /// Contains the error value
    Err(E),
}
```

The `Error` trait is, first and foremost, a way to **semantically** mark our type as being an error. It helps a reader of our codebase to immediately spot its purpose.
It is also a way for the Rust community to standardise on the minimum requirements for a **good** error:

- it should provide different representations (`Debug` and `Display`), tuned to different audiences;
- it should be possible to look at the underlying cause of the error, if any (`source`).

This list is still evolving - e.g. there is an unstable `backtrace` method.
Error handling is an active area of research in the Rust community - if you are interested in staying on top of what is coming next I strongly suggest you to keep an eye on the Rust Error Handling Working Group.

By providing a good implementation of all the optional methods we can fully leverage the error handling ecosystem - functions that have been designed to work with errors, generically. We will be writing one in a couple of sections!

8.2.2.1 Trait Objects

Before we work on implementing `source`, let's take a closer look at its return - `Option<&(dyn Error + 'static)>`.
`dyn Error` is a trait object[5] - a type that we know nothing about apart from the fact that it implements the `Error` trait.
Trait objects, just like generic type parameters, are a way to achieve polymorphism in Rust: invoke different implementations of the same interface. Generic types are resolved at compile-time (static dispatch), trait objects incur a runtime cost (dynamic dispatch).

Why does the standard library return a trait object?
It gives developers a way to access the underlying root cause of current error while keeping it *opaque*.
It does not leak any information about the type of the underlying root cause - you only get access to the methods exposed by the `Error` trait[6]: different representations (`Debug`, `Display`), the chance to go one level deeper in the **error chain** using `source`.

8.2.2.2 `Error::source`

Let's implement `Error` for `StoreTokenError`:

```
//! src/routes/subscriptions.rs
// [..]

impl std::error::Error for StoreTokenError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        // The compiler transparently casts `&sqlx::Error` into a `&dyn Error`
```

[5] Check out the relevant chapter in the Rust book for an in-depth introduction to trait objects.

[6] The `Error` trait provides a `downcast_ref` which can be used to obtain a concrete type back from `dyn Error`, assuming you know what type to downcast to. There are legitimate usecases for downcasting, but if you find yourself reaching for it too often it might be a sign that something is not quite right in your design/error handling strategy.

```rust
        Some(&self.0)
    }
}
```

`source` is useful when writing code that needs to handle a variety of errors: it provides a structured way to navigate the error chain without having to know anything about the specific error type you are working with.

If we look at our log record, the causal relationship between `StoreTokenError` and `sqlx::Error` is somewhat implicit - we infer one is the cause of the other because *it is a part of it*.

```
...
 INFO: [HTTP REQUEST - END]
    exception.details= StoreTokenError(
        Database(
            PgDatabaseError {
                severity: Error,
                code: "42703",
                message:
                    "column 'subscription_token' of relation
                     'subscription_tokens' does not exist",
                 ...
            }
        )
    )
    exception.message=
        "A database failure was encountered while
         trying to store a subscription token.",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

Let's go for something more explicit:

```rust
//! src/routes/subscriptions.rs

// Notice that we have removed `#[derive(Debug)]`
pub struct StoreTokenError(sqlx::Error);

impl std::fmt::Debug for StoreTokenError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "{}\nCaused by:\n\t{}", self, self.0)
    }
}
```

The log record leaves nothing to the imagination now:

```
...
 INFO: [HTTP REQUEST - END]
    exception.details=
        "A database failure was encountered
        while trying to store a subscription token.
        Caused by:
            error returned from database: column 'subscription_token'
            of relation 'subscription_tokens' does not exist"
    exception.message=
        "A database failure was encountered while
         trying to store a subscription token.",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

`exception.details` is easier to read and still conveys all the relevant information we had there before.

Using `source` we can write a function that provides a similar representation for any type that implements `Error`:

```rust
//! src/routes/subscriptions.rs
// [...]

fn error_chain_fmt(
    e: &impl std::error::Error,
    f: &mut std::fmt::Formatter<'_>,
) -> std::fmt::Result {
    writeln!(f, "{}\n", e)?;
    let mut current = e.source();
    while let Some(cause) = current {
        writeln!(f, "Caused by:\n\t{}", cause)?;
        current = cause.source();
    }
    Ok(())
}
```

It iterates over the whole chain of errors[7] that led to the failure we are trying to print.
We can then change our implementation of `Debug` for `StoreTokenError` to use it:

```rust
//! src/routes/subscriptions.rs
// [...]

impl std::fmt::Debug for StoreTokenError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        error_chain_fmt(self, f)
    }
}
```

The result is identical - and we can reuse it when working with other errors if we want a similar `Debug` representation.

8.3 Errors For Control Flow

8.3.1 Layering

We achieved the outcome we wanted (useful logs), but I am not too fond of the solution: we implemented a trait from our web framework (`ResponseError`) for an error type returned by an

[7] There is a `chain` method on `Error` that fulfills the same purpose - it has not been stabilised yet.

operation that is blissfully unaware of REST or the HTTP protocol, `store_token`. We could be calling `store_token` from a different entrypoint (e.g. a CLI) - nothing should have to change in its implementation.
Even assuming we are only ever going to be invoking `store_token` in the context of a REST API, we might add other endpoints that rely on that routine - they might not want to return a `500` when it fails.

Choosing the appropriate HTTP status code when an error occurs is a concern of the request handler, it should not leak elsewhere.
Let's delete

```rust
//! src/routes/subscriptions.rs
// [...]

// Nuke it!
impl ResponseError for StoreTokenError {}
```

To enforce a proper separation of concerns we need to introduce another error type, `SubscribeError`. We will use it as failure variant for `subscribe` and it will own the HTTP-related logic (`ResponseError`'s implementation).

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    // [...]
}

#[derive(Debug)]
struct SubscribeError {}

impl std::fmt::Display for SubscribeError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(
            f,
            "Failed to create a new subscriber."
        )
    }
}

impl std::error::Error for SubscribeError {}

impl ResponseError for SubscribeError {}
```

If you run `cargo check` you will see an avalanche of `'?' couldn't convert the error to 'SubscribeError'` - we need to implement conversions from the error types returned by our functions and `SubscribeError`.

8.3.2 Modelling Errors as Enums

An enum is the most common approach to work around this issue: a variant for each error type we need to deal with.

```rust
//! src/routes/subscriptions.rs
// [...]

#[derive(Debug)]
pub enum SubscribeError {
    ValidationError(String),
    DatabaseError(sqlx::Error),
    StoreTokenError(StoreTokenError),
    SendEmailError(reqwest::Error),
}
```

We can then leverage the `?` operator in our handler by providing a `From` implementation for each of wrapped error types:

```rust
//! src/routes/subscriptions.rs
// [...]

impl From<reqwest::Error> for SubscribeError {
    fn from(e: reqwest::Error) -> Self {
        Self::SendEmailError(e)
    }
}

impl From<sqlx::Error> for SubscribeError {
    fn from(e: sqlx::Error) -> Self {
        Self::DatabaseError(e)
    }
}

impl From<StoreTokenError> for SubscribeError {
    fn from(e: StoreTokenError) -> Self {
        Self::StoreTokenError(e)
    }
}

impl From<String> for SubscribeError {
    fn from(e: String) -> Self {
        Self::ValidationError(e)
    }
}
```

We can now clean up our request handler by removing all those `match` / `if fallible_function().is_err()` lines:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    let new_subscriber = form.0.try_into()?;
    let mut transaction = pool.begin().await?;
    let subscriber_id = insert_subscriber(/* */).await?;
    let subscription_token = generate_subscription_token();
    store_token(/* */).await?;
    transaction.commit().await?;
    send_confirmation_email(/* */).await?;
    Ok(HttpResponse::Ok().finish())
}
```

The code compiles, but one of our tests is failing:

```
thread 'subscriptions::subscribe_returns_a_400_when_fields_are_present_but_invalid'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `500`: The API did not return a 400 Bad Request when the payload was empty name.'
```

We are still using the default implementation of `ResponseError` - it always returns `500`.
This is where enums shine: we can use a `match` statement for **control flow** - we behave differently depending on the failure scenario we are dealing with.

```rust
//! src/routes/subscriptions.rs
use actix_web::http::StatusCode;
// [...]

impl ResponseError for SubscribeError {
    fn status_code(&self) -> StatusCode {
        match self {
            SubscribeError::ValidationError(_) => StatusCode::BAD_REQUEST,
            SubscribeError::DatabaseError(_)
              | SubscribeError::StoreTokenError(_)
              | SubscribeError::SendEmailError(_) => StatusCode::INTERNAL_SERVER_ERROR,
        }
    }
}
```

The test suite should pass again.

8.3.3 The Error Type Is Not Enough

What about our logs?
Let's look again:

```
export RUST_LOG="sqlx=error,info"
export TEST_LOG=enabled
cargo t subscribe_fails_if_there_is_a_fatal_database_error | bunyan
```

```
...
 INFO: [HTTP REQUEST - END]
    exception.details="StoreTokenError(
            A database failure was encountered while trying to
            store a subscription token.

        Caused by:
            error returned from database: column 'subscription_token'
            of relation 'subscription_tokens' does not exist)"
    exception.message="Failed to create a new subscriber.",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

We are still getting a great representation for the underlying `StoreTokenError` in `exception.details`, but it shows that we are now using the derived `Debug` implementation for `SubscribeError`. No loss of information though.
The same cannot be said for `exception.message` - no matter the failure mode, we always get

`Failed to create a new subscriber`. Not very useful.

Let's refine our `Debug` and `Display` implementations:

```rust
//! src/routes/subscriptions.rs
// [...]

// Remember to delete `#[derive(Debug)]`!
impl std::fmt::Debug for SubscribeError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        error_chain_fmt(self, f)
    }
}

impl std::error::Error for SubscribeError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match self {
            // &str does not implement `Error` - we consider it the root cause
            SubscribeError::ValidationError(_) => None,
            SubscribeError::DatabaseError(e) => Some(e),
            SubscribeError::StoreTokenError(e) => Some(e),
            SubscribeError::SendEmailError(e) => Some(e),
        }
    }
}

impl std::fmt::Display for SubscribeError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            SubscribeError::ValidationError(e) => write!(f, "{}", e),
            // What should we do here?
            SubscribeError::DatabaseError(_) => write!(f, "???"),
            SubscribeError::StoreTokenError(_) => write!(
                f,
                "Failed to store the confirmation token for a new subscriber."
            ),
            SubscribeError::SendEmailError(_) => {
                write!(f, "Failed to send a confirmation email.")
            },
        }
    }
}
```

`Debug` is easily sorted: we implemented the `Error` trait for `SubscribeError`, including `source`, and we can use again the helper function we wrote earlier for `StoreTokenError`.

We have a problem when it comes to `Display` - the same `DatabaseError` variant is used for errors encountered when:

- acquiring a new Postgres connection from the pool;
- inserting a subscriber in the `subscribers` table;
- committing the SQL transaction.

When implementing `Display` for `SubscribeError` we have no way to distinguish which of those three cases we are dealing with - the **underlying error type is not enough**.
Let's disambiguate by using a different enum variant for each operation:

```rust
//! src/routes/subscriptions.rs
// [...]

pub enum SubscribeError {
    // [...]
    // No more `DatabaseError`
    PoolError(sqlx::Error),
    InsertSubscriberError(sqlx::Error),
    TransactionCommitError(sqlx::Error),
}

impl std::fmt::Display for SubscribeError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            // [...]
            SubscribeError::PoolError(_) => {
                write!(f, "Failed to acquire a Postgres connection from the pool")
            }
            SubscribeError::InsertSubscriberError(_) => {
                write!(f, "Failed to insert new subscriber in the database.")
            }
            SubscribeError::TransactionCommitError(_) => {
                write!(
                    f,
                    "Failed to commit SQL transaction to store a new subscriber."
                )
            }
        }
    }
}

impl std::error::Error for SubscribeError {
    fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
        match self {
            //  [...]
            // No more DatabaseError
            SubscribeError::PoolError(e) => Some(e),
            SubscribeError::InsertSubscriberError(e) => Some(e),
            SubscribeError::TransactionCommitError(e) => Some(e),
            // [...]
        }
    }
}

impl ResponseError for SubscribeError {
    fn status_code(&self) -> StatusCode {
        match self {
            SubscribeError::ValidationError(_) => StatusCode::BAD_REQUEST,
            SubscribeError::PoolError(_)
            | SubscribeError::TransactionCommitError(_)
            | SubscribeError::InsertSubscriberError(_)
            | SubscribeError::StoreTokenError(_)
            | SubscribeError::SendEmailError(_) => StatusCode::INTERNAL_SERVER_ERROR,
        }
    }
}
```

`DatabaseError` is used in one more place:

```rust
//! src/routes/subscriptions.rs
// [..]

impl From<sqlx::Error> for SubscribeError {
    fn from(e: sqlx::Error) -> Self {
        Self::DatabaseError(e)
    }
}
```

The type alone is not enough to distinguish which of the new variants should be used; we cannot implement `From` for `sqlx::Error`.
We have to use `map_err` to perform the right conversion in each case.

```rust
//! src/routes/subscriptions.rs
// [..]

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    // [...]
    let mut transaction = pool.begin().await.map_err(SubscribeError::PoolError)?;
    let subscriber_id = insert_subscriber(&mut transaction, &new_subscriber)
        .await
        .map_err(SubscribeError::InsertSubscriberError)?;
    // [...]
    transaction
        .commit()
        .await
        .map_err(SubscribeError::TransactionCommitError)?;
    // [...]
}
```

The code compiles and `exception.message` is useful again:

```
...
 INFO: [HTTP REQUEST - END]
    exception.details="Failed to store the confirmation token
        for a new subscriber.

        Caused by:
            A database failure was encountered while trying to store
            a subscription token.
        Caused by:
            error returned from database: column 'subscription_token'
            of relation 'subscription_tokens' does not exist"
  exception.message="Failed to store the confirmation token for a new subscriber.",
    target=tracing_actix_web::root_span_builder,
    http.status_code=500
```

8.3.4 Removing The Boilerplate With `thiserror`

It took us roughly 90 lines of code to implement `SubscribeError` and all the machinery that surrounds it in order to achieve the desired behaviour and get useful diagnostic in our logs.
That is a *lot* of code, with a ton of boilerplate (e.g. `source`'s or `From` implementations).

Can we do better?

Well, I am not sure we can write less code, but we can find a different way out: we can **generate** all that boilerplate using a macro!

As it happens, there is already a great crate in the ecosystem for this purpose: `thiserror`. Let's add it to our dependencies:

```
#! Cargo.toml

[dependencies]
# [...]
thiserror = "1"
```

It provides a derive macro to generate most of the code we just wrote by hand.
Let's see it in action:

```rust
//! src/routes/subscriptions.rs
// [...]

#[derive(thiserror::Error)]
pub enum SubscribeError {
    #[error("{0}")]
    ValidationError(String),
    #[error("Failed to acquire a Postgres connection from the pool")]
    PoolError(#[source] sqlx::Error),
    #[error("Failed to insert new subscriber in the database.")]
    InsertSubscriberError(#[source] sqlx::Error),
    #[error("Failed to store the confirmation token for a new subscriber.")]
    StoreTokenError(#[from] StoreTokenError),
    #[error("Failed to commit SQL transaction to store a new subscriber.")]
    TransactionCommitError(#[source] sqlx::Error),
    #[error("Failed to send a confirmation email.")]
    SendEmailError(#[from] reqwest::Error),
}

// We are still using a bespoke implementation of `Debug`
// to get a nice report using the error source chain
impl std::fmt::Debug for SubscribeError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        error_chain_fmt(self, f)
    }
}

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    // We no longer have `#[from]` for `ValidationError`, so we need to
    // map the error explicitly
    let new_subscriber = form.0.try_into().map_err(SubscribeError::ValidationError)?;
    // [...]
}
```

We cut it down to 21 lines - not bad!
Let's break down what is happening now.

`thiserror::Error` is a procedural macro used via a `#[derive(/* */)]` attribute.
We have seen and used these before - e.g. `#[derive(Debug)]` or `#[derive(serde::Serialize)]`.

The macro receives, at compile-time, the definition of `SubscribeError` as input and returns another stream of tokens as output - it *generates new Rust code*, which is then compiled into the final binary.

Within the context of `#[derive(thiserror::Error)]` we get access to other attributes to achieve the behaviour we are looking for:

- `#[error(/* */)]` defines the `Display` representation of the enum variant it is applied to. E.g. `Display` will return `Failed to send a confirmation email.` when invoked on an instance of `SubscribeError::SendEmailError`. You can interpolate values in the final representation - e.g. the `{0}` in `#[error("{0}")]` on top of `ValidationError` is referring to the wrapped `String` field, mimicking the syntax to access fields on tuple structs (i.e. `self.0`).
- `#[source]` is used to denote what should be returned as root cause in `Error::source`;
- `#[from]` automatically derives an implementation of `From` for the type it has been applied to into the top-level error type (e.g. `impl From<StoreTokenError> for SubscribeError {/* */}`). The field annotated with `#[from]` is also used as error source, saving us from having to use two annotations on the same field (e.g. `#[source] #[from] reqwest::Error`).

I want to call your attention on a small detail: we are not using either `#[from]` or `#[source]` for the `ValidationError` variant. That is because `String` does not implement the `Error` trait, therefore it cannot be returned in `Error::source` - the same limitation we encountered before when implementing `Error::source` manually, which led us to return `None` in the `ValidationError` case.

8.4 Avoid "Ball Of Mud" Error Enums

In `SubscribeError` we are using enum variants for two purposes:

- Determine the response that should be returned to the caller of our API (`ResponseError`);
- Provide relevant diagnostic (`Error::source`, `Debug`, `Display`).

`SubscribeError`, as currently defined, exposes a lot of the implementation details of `subscribe`: we have a variant for every fallible function call we make in the request handler!
It is not a strategy that scales very well.

We need to think in terms of **abstraction layers**: what does a caller of `subscribe` **need** to know?

They should be able to determine what response to return to a user (via `ResponseError`). That's it.
The caller of `subscribe` does not understand the intricacies of the subscription flow: they don't know enough about the domain to behave differently for a `SendEmailError` compared to a `TransactionCommitError` (by design!). `subscribe` should return an error type that speaks at the **right level of abstraction**.

The ideal error type would look like this:

```rust
//! src/routes/subscriptions.rs

#[derive(thiserror::Error)]
pub enum SubscribeError {
    #[error("{0}")]
    ValidationError(String),
    #[error(/* */)]
    UnexpectedError(/* */),
}
```

`ValidationError` maps to a `400 Bad Request`, `UnexpectedError` maps to an opaque `500 Internal Server Error`.

What should we store in the `UnexpectedError` variant?
We need to map **multiple** error types into it - `sqlx::Error`, `StoreTokenError`, `reqwest::Error`.
We do not want to expose the implementation details of the fallible routines that get mapped to `UnexpectedError` by `subscribe` - it must be **opaque**.

We bumped into a type that fulfills those requirements when looking at the `Error` trait from Rust's standard library: `Box<dyn std::error::Error>`![8]

Let's give it a go:

```rust
//! src/routes/subscriptions.rs

#[derive(thiserror::Error)]
pub enum SubscribeError {
    #[error("{0}")]
    ValidationError(String),
    // Transparent delegates both `Display`'s and `source`'s implementation
    // to the type wrapped by `UnexpectedError`.
    #[error(transparent)]
    UnexpectedError(#[from] Box<dyn std::error::Error>),
}
```

We can still generate an accurate response for the caller:

```rust
//! src/routes/subscriptions.rs
// [...]

impl ResponseError for SubscribeError {
    fn status_code(&self) -> StatusCode {
        match self {
            SubscribeError::ValidationError(_) => StatusCode::BAD_REQUEST,
            SubscribeError::UnexpectedError(_) => StatusCode::INTERNAL_SERVER_ERROR,
        }
    }
}
```

We just need to adapt `subscribe` to properly convert our errors before using the `?` operator:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    // [...]
```

[8] We are wrapping `dyn std::error::Error` into a `Box` because the size of trait objects is not known at compile-time: trait objects can be used to store different types which will most likely have a different layout in memory. To use Rust's terminology, they are *unsized* - they do not implement the `Sized` marker trait. A `Box` stores the trait object itself on the heap, while we store the pointer to its heap location in `SubscribeError::UnexpectedError` - the pointer itself has a known size at compile-time - problem solved, we are `Sized` again.

```rust
    let mut transaction = pool
        .begin()
        .await
        .map_err(|e| SubscribeError::UnexpectedError(Box::new(e)))?;
    let subscriber_id = insert_subscriber(/* */)
        .await
        .map_err(|e| SubscribeError::UnexpectedError(Box::new(e)))?;
    // [...]
    store_token(/* */)
        .await
        .map_err(|e| SubscribeError::UnexpectedError(Box::new(e)))?;
    transaction
        .commit()
        .await
        .map_err(|e| SubscribeError::UnexpectedError(Box::new(e)))?;
    send_confirmation_email(/* */)
        .await
        .map_err(|e| SubscribeError::UnexpectedError(Box::new(e)))?;
    // [...]
}
```

There is some code repetition, but let it be for now.
The code compiles and our tests pass as expected.

Let's change the test we have used so far to check the quality of our log messages: let's trigger a failure in `insert_subscriber` instead of `store_token`.

```rust
//! tests/api/subscriptions.rs
// [...]

#[tokio::test]
async fn subscribe_fails_if_there_is_a_fatal_database_error() {
    // [...]
    // Break `subscriptions` instead of `subscription_tokens`
    sqlx::query!("ALTER TABLE subscriptions DROP COLUMN email;",)
        .execute(&app.db_pool)
        .await
        .unwrap();

    // [..]
}
```

The test passes, but we can see that our logs have regressed:

```
 INFO: [HTTP REQUEST - END]
    exception.details:
        "error returned from database: column 'email' of
         relation 'subscriptions' does not exist"
    exception.message:
        "error returned from database: column 'email' of
         relation 'subscriptions' does not exist"
```

We do not see a cause chain anymore.
We lost the operator-friendly error message that was previously attached to the `InsertSubscriberError` via `thiserror`:

```rust
//! src/routes/subscriptions.rs
// [...]

#[derive(thiserror::Error)]
pub enum SubscribeError {
    #[error("Failed to insert new subscriber in the database.")]
    InsertSubscriberError(#[source] sqlx::Error),
    // [...]
}
```

That is to be expected: we are forwarding the raw error now to `Display` (via `#[error(transparent)]`), we are not attaching any **additional context** to it in `subscribe`.
We can fix it - let’s add a new `String` field to `UnexpectedError` to attach contextual information to the opaque error we are storing:

```rust
//! src/routes/subscriptions.rs
// [...]

#[derive(thiserror::Error)]
pub enum SubscribeError {
    #[error("{0}")]
    ValidationError(String),
    #[error("{1}")]
    UnexpectedError(#[source] Box<dyn std::error::Error>, String),
}

impl ResponseError for SubscribeError {
    fn status_code(&self) -> StatusCode {
        match self {
            // [...]
            // The variant now has two fields, we need an extra `_`
            SubscribeError::UnexpectedError(_, _) => StatusCode::INTERNAL_SERVER_ERROR,
        }
    }
}
```

We need to adjust our mapping code in `subscribe` accordingly - we will reuse the error descriptions we had before refactoring `SubscribeError`:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    // [..]
    let mut transaction = pool.begin().await.map_err(|e| {
        SubscribeError::UnexpectedError(
            Box::new(e),
            "Failed to acquire a Postgres connection from the pool".into(),
        )
    })?;
    let subscriber_id = insert_subscriber(/* */)
        .await
        .map_err(|e| {
            SubscribeError::UnexpectedError(
                Box::new(e),
                "Failed to insert new subscriber in the database.".into(),
            )
        })?;
    // [..]
    store_token(/* */)
        .await
        .map_err(|e| {
            SubscribeError::UnexpectedError(
                Box::new(e),
                "Failed to store the confirmation token for a new subscriber.".into(),
            )
        })?;
    transaction.commit().await.map_err(|e| {
        SubscribeError::UnexpectedError(
            Box::new(e),
            "Failed to commit SQL transaction to store a new subscriber.".into(),
        )
    })?;
    send_confirmation_email(/* */)
        .await
        .map_err(|e| {
            SubscribeError::UnexpectedError(
                Box::new(e),
                "Failed to send a confirmation email.".into()
            )
        })?;
    // [..]
}
```

It is somewhat ugly, but it works:

```
INFO: [HTTP REQUEST - END]
   exception.details=
       "Failed to insert new subscriber in the database.

       Caused by:
           error returned from database: column 'email' of
           relation 'subscriptions' does not exist"
   exception.message="Failed to insert new subscriber in the database."
```

8.4.1 Using anyhow As Opaque Error Type

We could spend more time polishing the machinery we just built, but it turns out it is not necessary: we can lean on the ecosystem, again.
The author of thiserror[9] has another crate for us - anyhow.

```
#! Cargo.toml

[dependencies]
# [...]
anyhow = "1"
```

[9] It turns out that we are speaking of the same person that authored serde, syn, quote and many other foundational crates in the Rust ecosystem - @dtolnay. Consider sponsoring their OSS work.

The type we are looking for is `anyhow::Error`. Quoting the documentation:

> `anyhow::Error` is a wrapper around a dynamic error type. `anyhow::Error` works a lot like `Box<dyn std::error::Error>`, but with these differences:
>
> - `anyhow::Error` requires that the error is `Send`, `Sync`, and `'static`.
> - `anyhow::Error` guarantees that a backtrace is available, even if the underlying error type does not provide one.
> - `anyhow::Error` is represented as a narrow pointer — exactly one word in size instead of two.

The additional constraints (`Send`, `Sync` and `'static`) are not an issue for us.
We appreciate the more compact representation and the option to access a backtrace, if we were to be interested in it.

Let's replace `Box<dyn std::error::Error>` with `anyhow::Error` in `SubscribeError`:

```rust
//! src/routes/subscriptions.rs
// [...]

#[derive(thiserror::Error)]
pub enum SubscribeError {
    #[error("{0}")]
    ValidationError(String),
    #[error(transparent)]
    UnexpectedError(#[from] anyhow::Error),
}

impl ResponseError for SubscribeError {
    fn status_code(&self) -> StatusCode {
        match self {
            // [...]
            // Back to a single field
            SubscribeError::UnexpectedError(_) => StatusCode::INTERNAL_SERVER_ERROR,
        }
    }
}
```

We got rid of the second `String` field as well in `SubscribeError::UnexpectedError` - it is no longer necessary.
`anyhow::Error` provides the capability to enrich an error with **additional context** out of the box.

```rust
//! src/routes/subscriptions.rs
use anyhow::Context;
// [...]

pub async fn subscribe(/* */) -> Result<HttpResponse, SubscribeError> {
    // [...]
    let mut transaction = pool
        .begin()
        .await
        .context("Failed to acquire a Postgres connection from the pool")?;
    let subscriber_id = insert_subscriber(/* */)
        .await
        .context("Failed to insert new subscriber in the database.")?;
    // [..]
    store_token(/* */)
        .await
        .context("Failed to store the confirmation token for a new subscriber.")?;
    transaction
        .commit()
        .await
        .context("Failed to commit SQL transaction to store a new subscriber.")?;
    send_confirmation_email(/* */)
        .await
        .context("Failed to send a confirmation email.")?;
    // [...]
}
```

The `context` method is performing double duties here:

- it converts the error returned by our methods into an `anyhow::Error`;
- it enriches it with additional context around the intentions of the caller.

`context` is provided by the `Context` trait - `anyhow` implements it for `Result`[10], giving us access to a fluent API to easily work with fallible functions of all kinds.

8.4.2 `anyhow` Or `thiserror`?

We have covered a lot of ground - time to address a common Rust myth:

> `anyhow` is for applications, `thiserror` is for libraries.

It is not the right framing to discuss error handling.
You need to reason about **intent**.

Do you expect the caller to behave differently based on the failure mode they encountered?
Use an error enumeration, empower them to match on the different variants. Bring in `thiserror` to write less boilerplate.

Do you expect the caller to just give up when a failure occurs? Is their main concern reporting the error to an operator or a user?
Use an opaque error, do not give the caller *programmatic* access to the error inner details. Use `anyhow` or `eyre` if you find their API convenient.

The misunderstanding arises from the observation that most Rust libraries return an error enum instead of `Box<dyn std::error::Error>` (e.g. `sqlx::Error`).
Library authors cannot (or do not want to) make assumptions on the intent of their users. They steer away from being opinionated (to an extent) - enums give users more control, if they need it.
Freedom comes at a price - the interface is more complex, users need to sift through 10+ variants trying to figure out which (if any) deserve special handling.
Reason carefully about your usecase and the assumptions you can afford to make in order to design the most appropriate error type - sometimes `Box<dyn std::error::Error>` or `anyhow::Error` are the most appropriate choice, even for libraries.

[10] This is a common pattern in the Rust community, known as **extension trait**, to provide additional methods for types exposed by the standard library (or other common crates in the ecosystem).

8.5 Who Should Log Errors?

Let's look again at the logs emitted when a request fails.

```
# sqlx logs are a bit spammy, cutting them out to reduce noise
export RUST_LOG="sqlx=error,info"
export TEST_LOG=enabled
cargo t subscribe_fails_if_there_is_a_fatal_database_error | bunyan
```

There are three error-level log records:

- one emitted by our code in `insert_subscriber`

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber(/* */) -> Result<Uuid, sqlx::Error> {
    // [...]
    sqlx::query!(/* */)
        .execute(transaction)
        .await
        .map_err(|e| {
            tracing::error!("Failed to execute query: {:?}", e);
            e
        })?;
    // [...]
}
```

- one emitted by `actix_web` when converting `SubscribeError` into an `actix_web::Error`;
- one emitted by `tracing_actix_web::TracingLogger`, our telemetry middleware.

We do not need to see the same information three times - we are emitting unnecessary log records which, instead of helping, make it more confusing for operators to understand what is happening (are those logs reporting the same error? Am I dealing with three different errors?).

As a rule of thumb,

> errors should be logged when they are handled.

If your function is propagating the error upstream (e.g. using the `?` operator), it should **not** log the error. It can, if it makes sense, add more context to it.
If the error is propagated all the way up to the request handler, delegate logging to a dedicated middleware - `tracing_actix_web::TracingLogger` in our case.

The log record emitted by `actix_web` is going to be removed in the next release. Let's ignore it for now.

Let's review the `tracing::error` statements in our own code:

```rust
//! src/routes/subscriptions.rs
// [...]

pub async fn insert_subscriber(/* */) -> Result<Uuid, sqlx::Error> {
    // [...]
    sqlx::query!(/* */)
        .execute(transaction)
        .await
        .map_err(|e| {
            // This needs to go, we are propagating the error via `?`
            tracing::error!("Failed to execute query: {:?}", e);
            e
        })?;
    // [...]
}

pub async fn store_token(/* */) -> Result<(), StoreTokenError> {
    sqlx::query!(/* */)
        .execute(transaction)
        .await
        .map_err(|e| {
            // This needs to go, we are propagating the error via `?`
            tracing::error!("Failed to execute query: {:?}", e);
            StoreTokenError(e)
        })?;
    Ok(())
}
```

Check the logs again to confirm they look pristine.

8.6 Summary

We used this chapter to learn error handling patterns "the hard way" - building an ugly but working prototype first, refining it later using popular crates from the ecosystem.
You should now have:

- a solid grasp on the different purposes fulfilled by errors in an application;
- the most appropriate tools to fulfill them.

Internalise the mental model we discussed (`Location` as columns, `Purpose` as rows):

	Internal	At the edge
Control Flow	Types, methods, fields	Status codes
Reporting	Logs/traces	Response body

Practice what you learned: we worked on the `subscribe` request handler, tackle `confirm` as an exercise to verify your understanding of the concepts we covered. Improve the response returned to the user when validation of form data fails.
You can look at the code in the GitHub repository as a reference implementation.

Some of the themes we discussed in this chapter (e.g. layering and abstraction boundaries) will make another appearance when talking about the overall layout and structure of our application. Something to look forward to!

Chapter 9

Naive Newsletter Delivery

Our project is not yet a viable newsletter service: it cannot send out a new episode!

We will use this chapter to bootstrap newsletter delivery using a naive implementation.
It will be an opportunity to deepen our understanding of techniques we touched upon in previous chapters while building the foundation for tackling more advanced topics (e.g. authentication/authorization, fault tolerance).

9.1 User Stories Are Not Set In Stone

What are we trying to achieve, exactly?

We can go back to the user story we wrote down in Chapter 2:

> 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.

It looks simple, at least on the surface. The devil, as always, is in the details.
For example, in Chapter 7 we refined our domain model of a subscriber - we now have **confirmed** and **unconfirmed** subscribers.
Which ones should receive our newsletter issues?

That user story, as it stands, cannot help us - it was written before we started to make the distinction!

Make a habit of revisiting user stories throughout the lifecycle of a project.
When you spend time working on a problem you end up deepening your understanding of its **domain**. You often acquire a more precise **language** that can be used to refine earlier attempts of describing the desired functionality.

For this specific case: we only want newsletter issues to be sent to **confirmed** subscribers. Let's amend the user story accordingly:

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

9.2 Do Not Spam Unconfirmed Subscribers

We can get started by writing an integration test that specifies what should *not* happen: unconfirmed subscribers should not receive newsletter issues.

In Chapter 7 we selected Postmark as our email delivery service. If we are not calling Postmark, we are not sending an email out.
We can build on this fact to orchestrate a scenario that allows us to verify our business rule: if all subscribers are unconfirmed, no request is fired to Postmark when we publish a newsletter issue.

Let's translate that into code:

```rust
//! tests/api/main.rs
// [...]
// New test module!
mod newsletter;
```

```rust
//! tests/api/newsletter.rs
use crate::helpers::{spawn_app, TestApp};
use wiremock::matchers::{any, method, path};
use wiremock::{Mock, ResponseTemplate};

#[tokio::test]
async fn newsletters_are_not_delivered_to_unconfirmed_subscribers() {
    // Arrange
    let app = spawn_app().await;
    create_unconfirmed_subscriber(&app).await;

    Mock::given(any())
        .respond_with(ResponseTemplate::new(200))
        // We assert that no request is fired at Postmark!
        .expect(0)
        .mount(&app.email_server)
        .await;

    // Act

    // A sketch of the newsletter payload structure.
    // We might change it later on.
    let newsletter_request_body = serde_json::json!({
         "title": "Newsletter title",
         "content": {
             "text": "Newsletter body as plain text",
             "html": "<p>Newsletter body as HTML</p>",
         }
    });
    let response = reqwest::Client::new()
         .post(&format!("{}/newsletters", &app.address))
         .json(&newsletter_request_body)
         .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(response.status().as_u16(), 200);
    // Mock verifies on Drop that we haven't sent the newsletter email
}

/// Use the public API of the application under test to create
/// an unconfirmed subscriber.
async fn create_unconfirmed_subscriber(app: &TestApp) {
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    let _mock_guard = Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .named("Create unconfirmed subscriber")
        .expect(1)
        .mount_as_scoped(&app.email_server)
        .await;
    app.post_subscriptions(body.into())
        .await
        .error_for_status()
        .unwrap();
}
```

It fails, as expected:

```
thread 'newsletter::newsletters_are_not_delivered_to_unconfirmed_subscribers'
panicked at 'assertion failed: `(left == right)`
  left: `404`,
 right: `200`'
```

There is no handler in our API for `POST /newsletters`: `actix-web` returns a `404 Not Found` instead of the `200 OK` the test is expecting.

9.2.1 Set Up State Using The Public API

Let's take a moment to look at the `Arrange` section for the test we just wrote.
Our test scenario makes some assumptions on the **state** of our application: we need to have one subscriber and they must be unconfirmed.

Each test spins up a brand-new application running on top of an empty database.

```
let app = spawn_app().await;
```

How do we fill it up according to the test requirements?

We stay true to the black-box approach we described in Chapter 3: when possible, we drive the application state by calling its public API.
That is what we are doing in `create_unconfirmed_subscriber`:

```
//! tests/api/newsletter.rs
// [...]

async fn create_unconfirmed_subscriber(app: &TestApp) {
```

```rust
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    let _mock_guard = Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .named("Create unconfirmed subscriber")
        .expect(1)
        .mount_as_scoped(&app.email_server)
        .await;
    app.post_subscriptions(body.into())
        .await
        .error_for_status()
        .unwrap();
}
```

We use the API client we built as part of `TestApp` to make a `POST` call to the `/subscriptions` endpoint.

9.2.2 Scoped Mocks

We know that `POST /subscriptions` will send a confirmation email out - we must make sure that our Postmark test server is ready to handle the incoming request by setting up the appropriate `Mock`.
The matching logic overlaps what we have in the test function body: how do we make sure the two mocks don't end up stepping on each other's toes?

We use a **scoped** mock:

```rust
let _mock_guard = Mock::given(path("/email"))
    .and(method("POST"))
    .respond_with(ResponseTemplate::new(200))
    .named("Create unconfirmed subscriber")
    .expect(1)
    // We are not using `mount`!
    .mount_as_scoped(&app.email_server)
    .await;
```

With `mount`, the behaviour we specify remains active as long as the underlying `MockServer` is up and running.
With `mount_as_scoped`, instead, we get back a guard object - a `MockGuard`.

`MockGuard` has a custom `Drop` implementation: when it goes out of scope, `wiremock` instructs the underlying `MockServer` to stop honouring the specified mock behaviour. In other words, we stop returning `200` to `POST /email` at the end of `create_unconfirmed_subscriber`.
The mock behaviour needed for our test helper **stays local** to the test helper itself.

One more thing happens when a `MockGuard` is dropped - we **eagerly** check that expectations on the scoped mock are verified.
This creates a useful feedback loop to keep our test helpers clean and up-to-date.

We have already witnessed how black-box testing pushes us to write an API client for our own application to keep our tests concise.
Over time, you build more and more helper functions to drive the application state - just like we just did with `create_unconfirmed_subscriber`. These helpers rely on mocks but, as the applica-

tion evolves, some of those mocks end up no longer being necessary - a call gets removed, you stop using a certain provider, etc.
Eager evaluation of expectations for scoped mocks helps us to keep helper code in check and proactively clean up where possible.

9.2.3 Green Test

We can get the test to pass by providing a dummy implementation of `POST /newsletters`:

```rust
//! src/routes/mod.rs
// [...]
// New module!
mod newsletters;

pub use newsletters::*;
```

```rust
//! src/routes/newsletters.rs
use actix_web::HttpResponse;

// Dummy implementation
pub async fn publish_newsletter() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

```rust
//! src/startup.rs
// [...]
use crate::routes::{confirm, health_check, publish_newsletter, subscribe};

fn run(/* */) -> Result<Server, std::io::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            .wrap(TracingLogger::default())
            // Register the new handler!
            .route("/newsletters", web::post().to(publish_newsletter))
            // [...]
    })
    // [...]
}
```

`cargo test` is happy again.

9.3 All Confirmed Subscribers Receive New Issues

Let's write another integration test, this time for a subset of the happy case: if we have one confirmed subscriber, they receive an email with the new issue of the newsletter.

9.3.1 Composing Test Helpers

As in the previous test, we need to get the application state where we want it to be before executing the test logic - it calls for another helper function, this time to create a confirmed subscriber.
By slightly reworking `create_unconfirmed_subscriber` we can avoid duplication:

```rust
//! tests/api/newsletter.rs
// [...]

async fn create_unconfirmed_subscriber(app: &TestApp) -> ConfirmationLinks {
    let body = "name=le%20guin&email=ursula_le_guin%40gmail.com";

    let _mock_guard = Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .named("Create unconfirmed subscriber")
        .expect(1)
        .mount_as_scoped(&app.email_server)
        .await;
    app.post_subscriptions(body.into())
        .await
        .error_for_status()
        .unwrap();

    // We now inspect the requests received by the mock Postmark server
    // to retrieve the confirmation link and return it
    let email_request = &app
        .email_server
        .received_requests()
        .await
        .unwrap()
        .pop()
        .unwrap();
    app.get_confirmation_links(&email_request)
}

async fn create_confirmed_subscriber(app: &TestApp) {
    // We can then reuse the same helper and just add
    // an extra step to actually call the confirmation link!
    let confirmation_link = create_unconfirmed_subscriber(app).await;
    reqwest::get(confirmation_link.html)
        .await
        .unwrap()
        .error_for_status()
        .unwrap();
}
```

Nothing needs to change in our existing test and we can immediately leverage `create_confirmed_subscriber` in the new one:

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn newsletters_are_delivered_to_confirmed_subscribers() {
    // Arrange
    let app = spawn_app().await;
    create_confirmed_subscriber(&app).await;

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .expect(1)
        .mount(&app.email_server)
        .await;

    // Act
    let newsletter_request_body = serde_json::json!({
        "title": "Newsletter title",
        "content": {
            "text": "Newsletter body as plain text",
            "html": "<p>Newsletter body as HTML</p>",
        }
    });
    let response = reqwest::Client::new()
        .post(&format!("{}/newsletters", &app.address))
        .json(&newsletter_request_body)
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(response.status().as_u16(), 200);
    // Mock verifies on Drop that we have sent the newsletter email
}
```

It fails, as it should:

```
thread 'newsletter::newsletters_are_delivered_to_confirmed_subscribers' panicked at
Verifications failed:
- Mock #1.
        Expected range of matching incoming requests: == 1
        Number of matched incoming requests: 0
```

9.4 Implementation Strategy

We have more than enough tests to give us feedback now - let's kick off the implementation work!

We will start with a naive approach:

- Retrieve the newsletter issue details from the body of the incoming API call;
- Fetch the list of all confirmed subscribers from the database;
- Iterate through the whole list:
  - Get the subscriber email;
  - Send an email out via Postmark.

Let's do it!

9.5 Body Schema

What do we need to know about a newsletter in order to deliver it?

If we are striving to keep it as simple as possible:

- the title, to be used as email subject;
- the content, in HTML and pure text, to satisfy all email clients out there.

We can encode our requirements using structs that derive `serde::Deserialize`, just like we did in `POST /subscriptions` with `FormData`.

```rust
//! src/routes/newsletters.rs
// [...]

#[derive(serde::Deserialize)]
pub struct BodyData {
    title: String,
    content: Content
}

#[derive(serde::Deserialize)]
pub struct Content {
    html: String,
    text: String
}
```

`serde` does not have any issue with our nested layout given that all field types in `BodyData` implement `serde::Deserialize`. We can then use an `actix-web` extractor to parse `BodyData` out of the incoming request body. There is just one question to answer: what serialization format are we using?

For `POST /subscriptions`, given that we were dealing with HTML forms, we used `application/x-www-form-urlencoded` as `Content-Type`.
For `POST /newsletters` we are not tied to a form embedded in a web page: we will use JSON, a common choice when building REST APIs.
The corresponding extractor is `actix_web::web::Json`:

```rust
//! src/routes/newsletters.rs
// [...]
use actix_web::web;

// We are prefixing `body` with a `_` to avoid
// a compiler warning about unused arguments
pub async fn publish_newsletter(_body: web::Json<BodyData>) -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

9.5.1 Test Invalid Inputs

Trust but verify: let's add a new test case that throws invalid data at our `POST /newsletters` endpoint.

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn newsletters_returns_400_for_invalid_data() {
    // Arrange
    let app = spawn_app().await;
    let test_cases = vec![
        (
            serde_json::json!({
                "content": {
                    "text": "Newsletter body as plain text",
                    "html": "<p>Newsletter body as HTML</p>",
                }
            }),
            "missing title",
        ),
        (
            serde_json::json!({"title": "Newsletter!"}),
            "missing content",
        ),
    ];

    for (invalid_body, error_message) in test_cases {
        let response = reqwest::Client::new()
            .post(&format!("{}/newsletters", &app.address))
            .json(&invalid_body)
            .send()
            .await
            .expect("Failed to execute request.");

        // Assert
        assert_eq!(
            400,
            response.status().as_u16(),
            "The API did not fail with 400 Bad Request when the payload was {}.",
            error_message
        );
    }
}
```

The new test passes - you can add a few more cases if you want to.
Let's seize the occasion to refactor a bit and remove some code duplication - we can extract the logic to fire a request to `POST /newsletters` into a shared helper method on `TestApp`, as we did for `POST /subscriptions`:

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]
    pub async fn post_newsletters(&self, body: serde_json::Value) -> reqwest::Response {
        reqwest::Client::new()
            .post(&format!("{}/newsletters", &self.address))
            .json(&body)
            .send()
            .await
            .expect("Failed to execute request.")
    }
}
```

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn newsletters_are_not_delivered_to_unconfirmed_subscribers() {
    // [...]
    let response = app.post_newsletters(newsletter_request_body).await;
    // [...]
}

#[tokio::test]
async fn newsletters_are_delivered_to_confirmed_subscribers() {
    // [...]
    let response = app.post_newsletters(newsletter_request_body).await;
    // [...]
}

#[tokio::test]
async fn newsletters_returns_400_for_invalid_data() {
    // [...]
    for (invalid_body, error_message) in test_cases {
        let response = app.post_newsletters(invalid_body).await;
        // [...]
    }
}
```

9.6 Fetch Confirmed Subscribers List

We need to write a new query to retrieve the list of all confirmed subscribers.
A `WHERE` clause on the `status` column is enough to isolate the rows we care about:

```rust
//! src/routes/newsletters.rs
// [...]
use sqlx::PgPool;

struct ConfirmedSubscriber {
    email: String,
}

#[tracing::instrument(name = "Get confirmed subscribers", skip(pool))]
async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<ConfirmedSubscriber>, anyhow::Error> {
    let rows = sqlx::query_as!(
        ConfirmedSubscriber,
        r#"
        SELECT email
        FROM subscriptions
        WHERE status = 'confirmed'
        "#,
    )
    .fetch_all(pool)
    .await?;
    Ok(rows)
}
```

There is something new in there: we are using `sqlx::query_as!` instead of `sqlx::query!`.
`sqlx::query_as!` maps the retrieved rows to the type specified as its first argument, `ConfirmedSubscriber`, saving us a bunch of boilerplate.

Notice that `ConfirmedSubscriber` has a single field - `email`. We are minimising the amount of data we are fetching from the database, limiting our query to the columns we actually need to

send a newsletter out. Less work for the database, less data to move over the network.
It won't make a noticeable difference in this case, but it is a good practice to keep in mind when working on bigger applications with a heavier data footprint.

To leverage `get_confirmed_subscribers` in our handler we need a `PgPool` - we can extract one from the application state, just like we did in `POST /subscriptions`.

```rust
//! src/routes/newsletters.rs
// [...]

pub async fn publish_newsletter(
    _body: web::Json<BodyData>,
    pool: web::Data<PgPool>,
) -> HttpResponse {
    let _subscribers = get_confirmed_subscribers(&pool).await?;
    HttpResponse::Ok().finish()
}
```

The compiler is not happy:

```
21 |   ) -> HttpResponse {
   |  ___________________-
22 | |     let subscribers = get_confirmed_subscribers(&pool).await?;
   | |                       ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   | |                       cannot use the `?` operator in an async function
   | |                       that returns `actix_web::HttpResponse`
   | |
23 | |     HttpResponse::Ok().finish()
24 | | }
   | |__ this function should return `Result` or `Option` to accept `?`
```

SQL queries may fail and so does `get_confirmed_subscribers` - we need to change the return type of `publish_newsletter`.
We need to return a `Result` with an appropriate error type, just like we did in the last chapter:

```rust
//! src/routes/newsletters.rs
// [...]
use actix_web::ResponseError;
use sqlx::PgPool;
use crate::routes::error_chain_fmt;
use actix_web::http::StatusCode;

#[derive(thiserror::Error)]
pub enum PublishError {
    #[error(transparent)]
    UnexpectedError(#[from] anyhow::Error),
}

// Same logic to get the full error chain on `Debug`
impl std::fmt::Debug for PublishError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        error_chain_fmt(self, f)
    }
}

impl ResponseError for PublishError {
    fn status_code(&self) -> StatusCode {
        match self {
            PublishError::UnexpectedError(_) => StatusCode::INTERNAL_SERVER_ERROR,
        }
    }
}

pub async fn publish_newsletter(
    body: web::Json<BodyData>,
    pool: web::Data<PgPool>,
) -> Result<HttpResponse, PublishError> {
    let subscribers = get_confirmed_subscribers(&pool).await?;
    Ok(HttpResponse::Ok().finish())
}
```

Using what we learned in Chapter 8 it doesn't take that much to roll out a new error type!
Let me remark that we are future-proofing our code a bit: we modelled `PublishError` as an enumeration, but we only have one variant at the moment. A struct (or `actix_web::error::InternalError`) would have been more than enough for the time being.

`cargo check` should succeed now.

9.7 Send Newsletter Emails

Time to send those emails out!
We can leverage the `EmailClient` we wrote a few chapters ago - just like `PgPool`, it is already part of the application state and we can extract it using `web::Data`.

```rust
//! src/routes/newsletters.rs
// [...]
use crate::email_client::EmailClient;

pub async fn publish_newsletter(
    body: web::Json<BodyData>,
    pool: web::Data<PgPool>,
    // New argument!
    email_client: web::Data<EmailClient>,
) -> Result<HttpResponse, PublishError> {
    let subscribers = get_confirmed_subscribers(&pool).await?;
    for subscriber in subscribers {
        email_client
            .send_email(
                subscriber.email,
                &body.title,
                &body.content.html,
                &body.content.text,
            )
            .await?;
    }
    Ok(HttpResponse::Ok().finish())
}
```

It *almost* works:

```
error[E0308]: mismatched types
  --> src/routes/newsletters.rs
   |
48 |                  subscriber.email,
   |                  ^^^^^^^^^^^^^^^^
   |   expected struct `SubscriberEmail`,
   |   found struct `std::string::String`

error[E0277]: `?` couldn't convert the error to `PublishError`
  --> src/routes/newsletters.rs:53:19
   |
53 |              .await?;
   |                    ^
   |   the trait `From<reqwest::Error>`
   |   is not implemented for `PublishError`
```

9.7.1 context Vs with_context

We can quickly fix the second one

```rust
//! src/routes/newsletters.rs
// [...]
// Bring anyhow's extension trait into scope!
use anyhow::Context;

pub async fn publish_newsletter(/* */) -> Result<HttpResponse, PublishError> {
    // [...]
    for subscriber in subscribers {
        email_client
            .send_email(/* */)
            .await
            .with_context(|| {
                format!("Failed to send newsletter issue to {}", subscriber.email)
            })?;
    }
    // [...]
}
```

We are using a new method, `with_context`.

It is a close relative of `context`, the method we used extensively in Chapter 8 to convert the error variant of `Result` into `anyhow::Error` while enriching it with contextual information.

There is one key difference between the two: `with_context` is **lazy**.
It takes a closure as argument and the closure is only called in case of an error.

If the context you are adding is static - e.g. `context("Oh no!")` - they are equivalent.
If the context you are adding has a runtime cost, use `with_context` - you avoid paying for the error path when the fallible operation succeeds.

Let's look at our case, as an example: `format!` allocates memory on the heap to store its output string. Using `context`, we would be allocating that string every time we send an email out.
Using `with_context`, instead, we only invoke `format!` if email delivery fails.

9.8 Validation Of Stored Data

`cargo check` should return a single error now:

```
error[E0308]: mismatched types
  --> src/routes/newsletters.rs
   |
48 |                 subscriber.email,
   |                 ^^^^^^^^^^^^^^^^
   |
   |    expected struct `SubscriberEmail`,
   |    found struct `std::string::String`
```

We are not performing any validation on the data we retrieve from the database - `ConfirmedSubscriber::email` is of type `String`.
`EmailClient::send_email`, instead, expects a validated email address - a `SubscriberEmail` instance.

We can try the naive solution first - change `ConfirmedSubscriber::email` to be of type `SubscriberEmail`.

```rust
//! src/routes/newsletters.rs
// [...]
use crate::domain::SubscriberEmail;

struct ConfirmedSubscriber {
    email: SubscriberEmail,
}

#[tracing::instrument(name = "Get confirmed subscribers", skip(pool))]
async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<ConfirmedSubscriber>, anyhow::Error> {
    let rows = sqlx::query_as!(
        ConfirmedSubscriber,
        r#"
        SELECT email
        FROM subscriptions
        WHERE status = 'confirmed'
        "#,
    )
    .fetch_all(pool)
    .await?;
    Ok(rows)
}
```

```
error[E0308]: mismatched types
  --> src/routes/newsletters.rs
   |
69 |        let rows = sqlx::query_as!(
   |   ________________^
70 | |         ConfirmedSubscriber,
71 | |         r#"
72 | |         SELECT email
...  |
75 | |         "#,
76 | |     )
   | |_____^ expected struct `SubscriberEmail`,
              found struct `std::string::String`
```

`sqlx` doesn't like it - it does not know how to convert a `TEXT` column into `SubscriberEmail`. We could scan `sqlx`'s documentation for a way to implement support for custom type - a lot of trouble for a minor upside.

We can follow a similar approach to the one we deployed for our `POST /subscriptions` endpoint - we use two structs:

- one encodes the data layout we expect on the wire (`FormData`);
- the other one is built by parsing the raw representation, using our domain types (`NewSubscriber`).

For our query, it looks like this:

```rust
//! src/routes/newsletters.rs
// [...]

struct ConfirmedSubscriber {
    email: SubscriberEmail,
}

#[tracing::instrument(name = "Get confirmed subscribers", skip(pool))]
async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<ConfirmedSubscriber>, anyhow::Error> {
    // We only need `Row` to map the data coming out of this query.
    // Nesting its definition inside the function itself is a simple way
    // to clearly communicate this coupling (and to ensure it doesn't
    // get used elsewhere by mistake).
    struct Row {
        email: String,
    }

    let rows = sqlx::query_as!(
        Row,
        r#"
        SELECT email
        FROM subscriptions
        WHERE status = 'confirmed'
        "#,
    )
    .fetch_all(pool)
    .await?;
    // Map into the domain type
    let confirmed_subscribers = rows
        .into_iter()
        .map(|r| ConfirmedSubscriber {
            // Just panic if validation fails
            email: SubscriberEmail::parse(r.email).unwrap(),
        })
        .collect();
    Ok(confirmed_subscribers)
}
```

Is that `SubscriberEmail::parse(r.email).unwrap()` a good idea?

The emails of all new subscribers go through the validation logic in `SubscriberEmail::parse` - it was a big focus topic for us in Chapter 6.
You might argue, then, that all the emails stored in our database are necessarily valid - there is no need to account validation failures here. It is safe to just `unwrap` them all, knowing it will never panic.

This reasoning is sound assuming our software never changes. But we are optimising for high deployment frequency!

Data stored in our Postgres instance creates a **temporal coupling between old and new versions of our application**.
The emails we are retrieving from our database were marked as a valid by a previous version of our application. The current version might disagree.

We might discover, for example, that our email validation logic is too lenient - some invalid emails are slipping through the cracks, leading to issues when attempting to deliver newsletters. We implement a stricter validation routine, deploy the patched version and, suddenly, email delivery does not work at all!
`get_confirmed_subscribers` panics when processing stored emails that were previously considered valid, but no longer are.

What should we do, then?
Should we skip validation entirely when retrieving data from the database?

There is no one-size-fits-all answer.
You need to evaluate the issue on a case by case basis given the requirements of your domain.

Sometimes it is unacceptable to process invalid records - the routine should fail and an operator must intervene to rectify the corrupt records.
Sometimes we need to process all historical records (e.g. analytics) and we should make minimal assumptions about the data - `String` is our safest bet.

In our case, we can meet half-way: we can skip invalid emails when fetching the list of recipients for our next newsletter issue. We will emit a warning for every invalid address we find, allowing an operator to identify the issue and correct the stored records at a certain point in the future.

```rust
//! src/routes/newsletters.rs
// [...]

async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<ConfirmedSubscriber>, anyhow::Error> {
    // [...]

    // Map into the domain type
    let confirmed_subscribers = rows
        .into_iter()
        .filter_map(|r| match SubscriberEmail::parse(r.email) {
            Ok(email) => Some(ConfirmedSubscriber { email }),
            Err(error) => {
                tracing::warn!(
                    "A confirmed subscriber is using an invalid email address.\n{}.",
                    error
                );
                None
            }
        })
        .collect();
    Ok(confirmed_subscribers)
}
```

`filter_map` is a handy combinator - it returns a new iterator containing only the items for which our closure returned a `Some` variant.

9.8.1 Responsibility Boundaries

We could get away with this, but it is worth taking a moment to reflect on who is doing what here. Is `get_confirmed_subscriber` the most appropriate location to choose if we should skip or abort when encountering an invalid email address?
It feels like a business-level decision that would be better placed in `publish_newsletter`, the driving routine of our delivery workflow.

`get_confirmed_subscriber` should simply act as an adapter between our storage layer and our domain layer. It deals with the database-specific bits (i.e. the query) and the mapping logic, but it delegates to the caller the decision on what to do if the mapping or the query fail.

Let's refactor:

```rust
//! src/routes/newsletters.rs
// [...]

async fn get_confirmed_subscribers(
    pool: &PgPool,
    // We are returning a `Vec` of `Result`s in the happy case.
    // This allows the caller to bubble up errors due to network issues or other
    // transient failures using the `?` operator, while the compiler
    // forces them to handle the subtler mapping error.
    // See http://sled.rs/errors.html for a deep-dive about this technique.
) -> Result<Vec<Result<ConfirmedSubscriber, anyhow::Error>>, anyhow::Error> {
    // [...]

    let confirmed_subscribers = rows
        .into_iter()
        // No longer using `filter_map`!
        .map(|r| match SubscriberEmail::parse(r.email) {
            Ok(email) => Ok(ConfirmedSubscriber { email }),
            Err(error) => Err(anyhow::anyhow!(error)),
        })
        .collect();
    Ok(confirmed_subscribers)
}
```

We now get a compiler error at the calling site

```
error[E0609]: no field `email` on type `Result<ConfirmedSubscriber, anyhow::Error>`
  --> src/routes/newsletters.rs
   |
50 |                  subscriber.email,
   |
```

which we can immediately fix:

```rust
//! src/routes/newsletters.rs
// [...]

pub async fn publish_newsletter(/* */) -> Result<HttpResponse, PublishError> {
    let subscribers = get_confirmed_subscribers(&pool).await?;
    for subscriber in subscribers {
        // The compiler forces us to handle both the happy and unhappy case!
        match subscriber {
            Ok(subscriber) => {
                email_client
                    .send_email(
                        subscriber.email,
                        &body.title,
                        &body.content.html,
                        &body.content.text,
                    )
                    .await
                    .with_context(|| {
                        format!(
                            "Failed to send newsletter issue to {}",
                            subscriber.email
                        )
                    })?;
            }
            Err(error) => {
                tracing::warn!(
                    // We record the error chain as a structured field
                    // on the log record.
                    error.cause_chain = ?error,
                    // Using `\` to split a long string literal over
                    // two lines, without creating a `\n` character.
                    "Skipping a confirmed subscriber. \
                    Their stored contact details are invalid",
                );
            }
        }
    }
    Ok(HttpResponse::Ok().finish())
}
```

9.8.2 Follow The Compiler

The compiler is almost happy:

```
error[E0277]: `SubscriberEmail` doesn't implement `std::fmt::Display`
  --> src/routes/newsletters.rs:59:74
   |
59 |  format!("Failed to send newsletter issue to {}", subscriber.email)
   |                                                  ^^^^^^^^^^^^^^^^
   |   `SubscriberEmail` cannot be formatted with the default formatter
```

This is caused by our type change for `email` in `ConfirmedSubscriber`, from `String` to `SubscriberEmail`.
Let's implement `Display` for our new type:

```rust
//! src/domain/subscriber_email.rs
// [...]


impl std::fmt::Display for SubscriberEmail {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        // We just forward to the Display implementation of
        // the wrapped String.
        self.0.fmt(f)
    }
}
```

Progress! Different compiler error, this time from the borrow checker!

```
error[E0382]: borrow of partially moved value: `subscriber`
  --> src/routes/newsletters.rs
   |
52 |      subscriber.email,
   |      ---------------- value partially moved here
...
58 |  .with_context(|| {
   |                ^^ value borrowed here after partial move
59 |      format!("Failed to send newsletter issue to {}", subscriber.email)
   |                                                       ----------
                                  borrow occurs due to use in closure
```

We could just slap a `.clone()` on the first usage and call it a day.
But let's try to be sophisticated: do we really need to take ownership of `SubscriberEmail` in `EmailClient::send_email`?

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

pub async fn send_email(
    &self,
    recipient: SubscriberEmail,
    /* */
) -> Result<(), reqwest::Error> {
    // [...]
    let request_body = SendEmailRequest {
        to: recipient.as_ref(),
        // [...]
    };
    // [...]
}
```

We just need to be able to call `as_ref` on it - a `&SubscriberEmail` would work just fine.
Let's change the signature accordingly:

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

pub async fn send_email(
    &self,
    recipient: &SubscriberEmail,
    /* */
) -> Result<(), reqwest::Error> {
    // [...]
}
```

There are a few calling sites that need to be updated - the compiler is gentle enough to point them out. I'll leave the fixes to you, the reader, as an exercise.
The test suite should pass when you are done.

9.8.3 Remove Some Boilerplate

Before moving forward, let's take one last look at `get_confirmed_subscribers`:

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(name = "Get confirmed subscribers", skip(pool))]
async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<Result<ConfirmedSubscriber, anyhow::Error>>, anyhow::Error> {
    struct Row {
        email: String,
    }

    let rows = sqlx::query_as!(
        Row,
        r#"
        SELECT email
        FROM subscriptions
        WHERE status = 'confirmed'
        "#,
    )
    .fetch_all(pool)
    .await?;
    let confirmed_subscribers = rows
        .into_iter()
        .map(|r| match SubscriberEmail::parse(r.email) {
            Ok(email) => Ok(ConfirmedSubscriber { email }),
            Err(error) => Err(anyhow::anyhow!(error)),
        })
        .collect();
    Ok(confirmed_subscribers)
}
```

Is `Row` adding any value?
Not really - the query is simple enough that we do not benefit significantly from having a dedicated type to represent the returned data.
We can switch back to `query!` and remove `Row` entirely:

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(name = "Get confirmed subscribers", skip(pool))]
async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<Result<ConfirmedSubscriber, anyhow::Error>>, anyhow::Error> {
    let confirmed_subscribers = sqlx::query!(
        r#"
        SELECT email
        FROM subscriptions
        WHERE status = 'confirmed'
        "#,
    )
    .fetch_all(pool)
    .await?
    .into_iter()
    .map(|r| match SubscriberEmail::parse(r.email) {
        Ok(email) => Ok(ConfirmedSubscriber { email }),
        Err(error) => Err(anyhow::anyhow!(error)),
    })
    .collect();
    Ok(confirmed_subscribers)
}
```

We didn't even need to touch the remaining code - it compiled straight-away.

9.9 Limitations Of The Naive Approach

We did it - we have an implementation that passes our two integration tests!
What now? Do we pat ourselves on the back and ship it to production?

Not so fast.
We said it at the very beginning - the approach we took is the simplest possible to get something up and running.
Is it good enough, though?

Let's have a hard look at its shortcomings!

1. **Security**
   Our `POST /newsletters` endpoint is unprotected - anyone can fire a request to it and broadcast to our entire audience, unchecked.
2. **You Only Get One Shot**
   As soon you hit `POST /newsletters`, your content goes out to your entire mailing list. No chance to edit or review it in draft mode before giving the green light for publishing.
3. **Performance**
   We are sending emails out one at a time.
   We wait for the current one to be dispatched successfully before moving on to the next in line.
   This is not a massive issue if you have 10 or 20 subscribers, but it becomes noticeable shortly afterwards: latency is going to be horrible for newsletters with a sizeable audience.
4. **Fault Tolerance**
   If we fail to dispatch one email we bubble up the error using `?` and return a `500 Internal Server Error` to the caller.
   The remaining emails are never sent, nor we retry to dispatch the failed one.
5. **Retry Safety**
   Many things can go wrong when communicating over the network. What should a consumer of our API do if they experience a timeout or a `500 Internal Server Error` when calling our service?

They cannot retry - they risk sending the newsletter issue twice to the entire mailing list.

Number 2. and 3. are annoying, but we could live with them for a while.
Number 4. and 5. are fairly serious limitations, with a visible impact on our audience.
Number 1. is simply non-negotiable: we **must** protect the endpoint before releasing our API.

9.10 Summary

We built a prototype of our newsletter delivery logic: it satisfies our functional requirements, but it is not yet ready for prime time.
The shortcomings of our MVP will become the focus of the next chapters, in priority order: we will tackle authentication/authorization first before moving on to fault tolerance.

Chapter 10

Securing Our API

In Chapter 9 we added a new endpoint to our API - `POST /newsletters`.
It takes a newsletter issue as input and sends emails out to all our subscribers.

We have an issue though - anybody can hit the API and broadcast whatever they want to our entire mailing list.

It is time to level up our API security toolbox.
We will introduce the concepts of authentication and authorization, evaluate various approaches (Basic auth, session-based auth, OAuth 2.0, OpenId Connect) and discuss the benefits (and pitfalls) of one of the most used token formats, JSON Web Tokens (JWTs).

> This chapter, like others in the book, chooses to "do it wrong" first for teaching purposes.
> Make sure to read until the end if you don't want to pick up bad security habits!

10.1 Authentication

We need a way to verify **who** is calling `POST /newsletters`.
Only a handful of people, the ones in charge of the content, should be able to send emails out to the entire mailing list.

We need to find a way to verify the **identity** of API callers - we must **authenticate** them.
How?

By asking for something they are uniquely positioned to provide.
There are various approaches, but they all boil down to three categories:

1. Something they know (e.g. passwords, PINs, security questions);
2. Something they have (e.g. a smartphone, using an authenticator app);
3. Something they are (e.g. fingerprints, Apple's Face ID).

Each approach has its weaknesses.

10.1.1 Drawbacks

10.1.1.1 Something They Know

Passwords must be long - short ones are vulnerable to brute-force attacks.
Passwords must be unique - publicly available information (e.g. date of birth, names of family members, etc.) should not give an attacker any chance to "guess" a password.
Passwords should not be reused across multiple services - if any of them gets compromised you risk granting access to all the other services sharing the same password.

On average, a person has 100 or more online accounts - they cannot be asked to remember hundreds of long unique passwords by heart.
Password managers help, but they are not mainstream yet and the user experience is often suboptimal.

10.1.1.2 Something They Have

Smartphones and U2F keys can be lost, locking the user out of their accounts.
They can also be stolen or compromised, giving an attacker a window of opportunity to impersonate the victim.

10.1.1.3 Something They Are

Biometrics, unlike passwords, cannot be changed - you cannot "rotate" your fingerprint or change the pattern of your retina's blood vessel.
Forging a fingerprint turns out to be easier than most would imagine - it is also information often available to government agencies who might abuse it or lose it.

10.1.2 Multi-factor Authentication

What should we do then, given that each approach has its own flaws?
Well, we could combine them!

That is pretty much what **multi-factor authentication** (MFA) boils down to - it requires the user to provide at least two different types of authentication factors in order to get access.

10.2 Password-based Authentication

Let's jump from theory to practice: how do we **implement** authentication?

Passwords look like the simplest approach among the three we mentioned.
How should we pass a username and a password to our API?

10.2.1 Basic Authentication

We can use the 'Basic' Authentication Scheme, a standard defined by the Internet Engineering Task Force (IETF) in RFC 2617 and later updated by RFC 7617.

The API must look for the `Authorization` header in the incoming request, structured as follows:

```
Authorization: Basic <encoded credentials>
```

where `<encoded credentials>` is the base64-encoding of `{username}:{password}`[1].

According to the specification, we need to partition our API into protection spaces or **realms** - resources within the same realm are protected using the same authentication scheme and set of credentials.
We only have a single endpoint to protect - `POST /newsletters`. We will therefore have a single realm, named `publish`.

The API must reject all requests missing the header or using invalid credentials - the response must use the `401 Unauthorized` status code and include a special header, `WWW-Authenticate`, containing a **challenge**.
The challenge is a string explaining to the API caller what type of authentication scheme we expect to see for the relevant realm.
In our case, using basic authentication, it should be:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="publish"
```

Let's implement it!

10.2.1.1 Extracting Credentials

Extracting username and password from the incoming request will be our first milestone.

Let's start with an unhappy case - an incoming request without an `Authorization` header is rejected.

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn requests_missing_authorization_are_rejected() {
    // Arrange
    let app = spawn_app().await;

    let response = reqwest::Client::new()
        .post(&format!("{}/newsletters", &app.address))
        .json(&serde_json::json!({
            "title": "Newsletter title",
            "content": {
                "text": "Newsletter body as plain text",
                "html": "<p>Newsletter body as HTML</p>",
            }
        }))
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(401, response.status().as_u16());
    assert_eq!(r#"Basic realm="publish""#, response.headers()["WWW-Authenticate"]);
}
```

[1] base64-encoding ensures that all the characters in the output are ASCII, but it does not provide any kind of protection: decoding requires no secrets. In other words, encoding is not encryption!

It fails at the first assertion:

```
thread 'newsletter::requests_missing_authorization_are_rejected' panicked at
'assertion failed: `(left == right)`
  left: `401`,
 right: `400`'
```

We must update our handler to fulfill the new requirements.
We can use the `HttpRequest` extractor to reach the headers associated with the incoming request:

```rust
//! src/routes/newsletters.rs
// [...]
use secrecy::Secret;
use actix_web::HttpRequest;
use actix_web::http::header::{HeaderMap, HeaderValue};

pub async fn publish_newsletter(
    // [...]
    // New extractor!
    request: HttpRequest,
) -> Result<HttpResponse, PublishError> {
    let _credentials = basic_authentication(request.headers());
    // [...]
}

struct Credentials {
    username: String,
    password: Secret<String>,
}

fn basic_authentication(headers: &HeaderMap) -> Result<Credentials, anyhow::Error> {
    todo!()
}
```

To extract the credentials we will need to deal with the base64 encoding.
Let's add the `base64` crate as a dependency:

```toml
[dependencies]
# [...]
base64 = "0.13"
```

We can now write down the body of `basic_authentication`:

```rust
//! src/routes/newsletters.rs
// [...]

fn basic_authentication(headers: &HeaderMap) -> Result<Credentials, anyhow::Error> {
    // The header value, if present, must be a valid UTF8 string
    let header_value = headers
        .get("Authorization")
        .context("The 'Authorization' header was missing")?
        .to_str()
        .context("The 'Authorization' header was not a valid UTF8 string.")?;
    let base64encoded_segment = header_value
        .strip_prefix("Basic ")
        .context("The authorization scheme was not 'Basic'.")?;
    let decoded_bytes = base64::decode_config(base64encoded_segment, base64::STANDARD)
```

```rust
        .context("Failed to base64-decode 'Basic' credentials.")?;
    let decoded_credentials = String::from_utf8(decoded_bytes)
        .context("The decoded credential string is not valid UTF8.")?;

    // Split into two segments, using ':' as delimitator
    let mut credentials = decoded_credentials.splitn(2, ':');
    let username = credentials
        .next()
        .ok_or_else(|| anyhow::anyhow!("A username must be provided in 'Basic' auth."))?
        .to_string();
    let password = credentials
        .next()
        .ok_or_else(|| anyhow::anyhow!("A password must be provided in 'Basic' auth."))?
        .to_string();

    Ok(Credentials {
        username,
        password: Secret::new(password)
    })
}
```

Take a moment to go through the code, line by line, and fully understand what is happening. Many operations that could go wrong!
Having the RFC open, side to side with the book, helps!

We are not done yet - our test is still failing.
We need to act on the error returned by `basic_authentication`:

```rust
//! src/routes/newsletters.rs
// [...]

#[derive(thiserror::Error)]
pub enum PublishError {
    // New error variant!
    #[error("Authentication failed.")]
    AuthError(#[source] anyhow::Error),
    #[error(transparent)]
    UnexpectedError(#[from] anyhow::Error),
}

impl ResponseError for PublishError {
    fn status_code(&self) -> StatusCode {
        match self {
            PublishError::UnexpectedError(_) => StatusCode::INTERNAL_SERVER_ERROR,
            // Return a 401 for auth errors
            PublishError::AuthError(_) => StatusCode::UNAUTHORIZED,
        }
    }
}


pub async fn publish_newsletter(/* */) -> Result<HttpResponse, PublishError> {
    let _credentials = basic_authentication(request.headers())
        // Bubble up the error, performing the necessary conversion
        .map_err(PublishError::AuthError)?;
    // [...]
}
```

Our status code assertion is now happy, the header one not yet:

```
thread 'newsletter::requests_missing_authorization_are_rejected' panicked at
'no entry found for key "WWW-Authenticate"'
```

So far it has been enough to specify which status code to return for each error - now we need something more, a header.
We need to change our focus from `ResponseError::status_code` to `ResponseError::error_response`:

```rust
//! src/routes/newsletters.rs
// [...]
use actix_web::http::{StatusCode, header};
use actix_web::http::header::{HeaderMap, HeaderValue};

impl ResponseError for PublishError {
    fn error_response(&self) -> HttpResponse {
        match self {
            PublishError::UnexpectedError(_) => {
                HttpResponse::new(StatusCode::INTERNAL_SERVER_ERROR)
            }
            PublishError::AuthError(_) => {
                let mut response = HttpResponse::new(StatusCode::UNAUTHORIZED);
                let header_value = HeaderValue::from_str(r#"Basic realm="publish""#)
                    .unwrap();
                response
                    .headers_mut()
                    // actix_web::http::header provides a collection of constants
                    // for the names of several well-known/standard HTTP headers
                    .insert(header::WWW_AUTHENTICATE, header_value);
                response
            }
        }
    }

    // `status_code` is invoked by the default `error_response`
    // implementation. We are providing a bespoke `error_response` implementation
    // therefore there is no need to maintain a `status_code` implementation anymore.
}
```

Our authentication test passes!
A few of the old ones are broken though:

```
test newsletter::newsletters_are_not_delivered_to_unconfirmed_subscribers ... FAILED
test newsletter::newsletters_are_delivered_to_confirmed_subscribers ... FAILED

thread 'newsletter::newsletters_are_not_delivered_to_unconfirmed_subscribers'
panicked at 'assertion failed: `(left == right)`
  left: `401`,
 right: `200`'

thread 'newsletter::newsletters_are_delivered_to_confirmed_subscribers'
panicked at 'assertion failed: `(left == right)`
  left: `401`,
 right: `200`'
```

`POST /newsletters` is now rejecting all unauthenticated requests, including the ones we were making in our happy-path black-box tests.
We can stop the bleeding by providing a random combination of username and password:

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    pub async fn post_newsletters(&self, body: serde_json::Value) -> reqwest::Response {
        reqwest::Client::new()
            .post(&format!("{}/newsletters", &self.address))
            // Random credentials!
            // `reqwest` does all the encoding/formatting heavy-lifting for us.
            .basic_auth(Uuid::new_v4().to_string(), Some(Uuid::new_v4().to_string()))
            .json(&body)
            .send()
            .await
            .expect("Failed to execute request.")
    }

    // [...]
}
```

The test suite is green again.

10.2.2 Password Verification - Naive Approach

An authentication layer that accepts random credentials is... not ideal.
We need to start validating the credentials we are extracting from the `Authorization` header - they should be compared to a list of known users.

We will create a new `users` Postgres table to store this list:

```
sqlx migrate add create_users_table
```

A first draft for the schema might look like this:

```sql
-- migrations/20210815112026_create_users_table.sql
CREATE TABLE users(
   user_id uuid PRIMARY KEY,
   username TEXT NOT NULL UNIQUE,
   password TEXT NOT NULL
);
```

We can then update our handler to query it every time we perform authentication:

```rust
//! src/routes/newsletters.rs
use secrecy::ExposeSecret;
// [...]

async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let user_id: Option<_> = sqlx::query!(
        r#"
        SELECT user_id
        FROM users
        WHERE username = $1 AND password = $2
        "#,
        credentials.username,
        credentials.password.expose_secret()
    )
    .fetch_optional(pool)
    .await
    .context("Failed to perform a query to validate auth credentials.")
    .map_err(PublishError::UnexpectedError)?;

    user_id
        .map(|row| row.user_id)
        .ok_or_else(|| anyhow::anyhow!("Invalid username or password."))
        .map_err(PublishError::AuthError)
}

pub async fn publish_newsletter(/* */) -> Result<HttpResponse, PublishError> {
    let credentials = basic_authentication(request.headers())
        .map_err(PublishError::AuthError)?;
    let user_id = validate_credentials(credentials, &pool).await?;
    // [...]
}
```

It would be a good idea to record who is calling `POST /newsletters` - let's add a `tracing` span around our handler:

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(
    name = "Publish a newsletter issue",
    skip(body, pool, email_client, request),
    fields(username=tracing::field::Empty, user_id=tracing::field::Empty)
)]
pub async fn publish_newsletter(/* */) -> Result<HttpResponse, PublishError> {
    let credentials = basic_authentication(request.headers())
        .map_err(PublishError::AuthError)?;
    tracing::Span::current().record(
        "username",
        &tracing::field::display(&credentials.username)
    );
    let user_id = validate_credentials(credentials, &pool).await?;
    tracing::Span::current().record("user_id", &tracing::field::display(&user_id));
    // [...]
}
```

We now need to update our happy-path tests to specify a username-password pair that is accepted by `validate_credentials`.
We will generate a test user for every instance of our test application. We have not yet implemented a sign-up flow for newsletter editors, therefore we cannot go for a fully black-box approach - for the time being we will inject the test user details directly into the database:

```rust
//! tests/api/helpers.rs
// [...]
```

```rust
pub async fn spawn_app() -> TestApp {
    // [...]

    let test_app = TestApp {/* */};
    add_test_user(&test_app.db_pool).await;
    test_app
}


async fn add_test_user(pool: &PgPool) {
    sqlx::query!(
        "INSERT INTO users (user_id, username, password)
        VALUES ($1, $2, $3)",
        Uuid::new_v4(),
        Uuid::new_v4().to_string(),
        Uuid::new_v4().to_string(),
    )
    .execute(pool)
    .await
    .expect("Failed to create test users.");
}
```

`TestApp` will provide a helper method to retrieve its username and password

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]

    pub async fn test_user(&self) -> (String, String) {
        let row = sqlx::query!("SELECT username, password FROM users LIMIT 1",)
            .fetch_one(&self.db_pool)
            .await
            .expect("Failed to create test users.");
        (row.username, row.password)
    }
}
```

which we will then be calling from our `post_newsletters` method, instead of using random credentials:

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]

    pub async fn post_newsletters(&self, body: serde_json::Value) -> reqwest::Response {
        let (username, password) = self.test_user().await;
        reqwest::Client::new()
            .post(&format!("{}/newsletters", &self.address))
            // No longer randomly generated on the spot!
            .basic_auth(username, Some(password))
            .json(&body)
            .send()
            .await
            .expect("Failed to execute request.")
    }
}
```

All our tests are passing now.

10.2.3 Password Storage

Storing raw user passwords in your database is not a good idea.

An attacker with access to your stored data can immediately start impersonating your users - both usernames and passwords are ready to go.
They don't even have to compromise your live database - an unencrypted backup is enough.

10.2.3.1 No Need To Store Raw Passwords

Why are we even storing passwords in the first place?
We need to perform an *equality check* - every time a user tries to authenticate we verify that the password they provided matches the password we were expecting.

If equality is all we care about, we can start devising a more sophisticated strategy.
We could, for example, *transform* the passwords by applying a function before comparing them.

All deterministic functions return the same output given the same input.
Let `f` be our deterministic function: `psw_candidate == expected_psw` implies `f(psw_candidate) == f(expected_psw)`.
This is not enough though - what if `f` returned `hello` for every possible input string? Password verification would succeed no matter the input provided.

We need to go in the opposite direction: if `f(psw_candidate) == f(expected_psw)` **then** `psw_candidate == expected_psw`.
This is possible assuming that our function `f` has an additional property: it must be *injective* - if `x != y` then `f(x) != f(y)`.

If we had such a function `f`, we could avoid storing the raw password altogether: when a user signs up, we compute `f(password)` and store it in our database. `password` is discarded.
When the same user tries to sign in, we compute `f(psw_candidate)` and check that it matches the `f(password)` value we stored during sign-up. The raw password is never persisted.

Does this actually improve our security posture?
It depends on `f`!

It is not that difficult to define an injective function - the reverse function, `f("hello") = "olleh"`, satisfies our criteria. It is equally easy to guess how to *invert* the transformation to recover the original password - it doesn't hinder an attacker.
We could make the transformation a lot more complicated - complicated enough to make it cumbersome for an attacker to find the inverse transformation.
Even that might not be enough. It is often sufficient for an attacker to be able to recover some properties of the input (e.g. length) from the output to mount, for example, a targeted brute-force attack.
We need something stronger - there should be no relationship between how similar two inputs `x` and `y` are and how similar the corresponding outputs `f(x)` and `f(y)` are.

We want a **cryptographic hash function**.
Hash functions map strings from the input space to **fixed-length** outputs.
The adjective **cryptographic** refers to the uniformity property we were just discussing, also known as **avalanche effect**: a tiny difference in inputs leads to outputs so different to the point of looking uncorrelated.

There is a caveat: hash functions are not injective[2], there is a tiny risk of collisions - if `f(x) == f(y)` there is a high probability (not 100%!) that `x == y`.

10.2.3.2 Using A Cryptographic Hash

Enough with the theory - let's update our implementation to hash passwords before storing them.

There are several cryptographic hash functions out there - MD5, SHA-1, SHA-2, SHA-3, KangarooTwelve, etc.
We are not going to delve deep into the pros and cons of each algorithm - it is pointless when it comes to passwords, for reasons that will become clear in a few pages.
For the sake of this section, let's move forward with SHA-3, the latest addition to the Secure Hash Algorithms family.

On top of the algorithm, we also need to choose the **output size** - e.g. SHA3-224 uses the SHA-3 algorithm to produce a fixed-sized output of 224 bits.
The options are 224, 256, 384 and 512. The longer the output, the more unlikely we are to experience a collision. On the flip side, we will need more storage and consume more bandwidth by using longer hashes.
SHA3-256 should be more than enough for our usecase.

The Rust Crypto organization provides an implementation of SHA-3, the `sha3` crate. Let's add it to our dependencies:

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

[dependencies]
# [...]
sha3 = "0.9"
```

For clarity, let's rename our `password` column to `password_hash`:

```
sqlx migrate add rename_password_column
```

```
-- migrations/20210815112028_rename_password_column.sql
ALTER TABLE users RENAME password TO password_hash;
```

Our project should stop compiling:

```
error: error returned from database: column "password" does not exist
  --> src/routes/newsletters.rs
   |
90 |       let user_id: Option<_> = sqlx::query!(
   |  ______________________________^
```

[2] Assuming that the input space is finite (i.e. password length is capped), it is theoretically possible to find a perfect hash function - `f(x) == f(y)` implies `x == y`.

```
91 | |         r#"
92 | |         SELECT user_id
93 | |         FROM users
...  |
97 | |         credentials.password
98 | |     )
   | |_____^
```

`sqlx::query!` spotted that one of our queries is using a column that no longer exists in the current schema.
Compile-time verification of SQL queries is quite neat, isn't it?

Our `validate_credentials` function looks like this:

```rust
//! src/routes/newsletters.rs
//! [...]

async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let user_id: Option<_> = sqlx::query!(
        r#"
        SELECT user_id
        FROM users
        WHERE username = $1 AND password = $2
        "#,
        credentials.username,
        credentials.password.expose_secret()
    )
    // [...]
}
```

Let's update it to work with hashed passwords:

```rust
//! src/routes/newsletters.rs
//! [...]
use sha3::Digest;

async fn validate_credentials(/* */) -> Result<uuid::Uuid, PublishError> {
    let password_hash = sha3::Sha3_256::digest(
        credentials.password.expose_secret().as_bytes()
    );
    let user_id: Option<_> = sqlx::query!(
        r#"
        SELECT user_id
        FROM users
        WHERE username = $1 AND password_hash = $2
        "#,
        credentials.username,
        password_hash
    )
    // [...]
}
```

Unfortunately, it will not compile straight away:

```
error[E0308]: mismatched types
  --> src/routes/newsletters.rs:99:9
   |
99 |         password_hash
   |         ^^^^^^^^^^^^^ expected `&str`, found struct `GenericArray`
   |
   = note: expected reference `&str`
                 found struct `GenericArray<u8, UInt<..>>`
```

`Digest::digest` returns a fixed-length array of bytes, while our `password_hash` column is of type `TEXT`, a string.
We could change the schema of the `users` table to store `password_hash` as binary. Alternatively, we can encode the bytes returned by `Digest::digest` in a string using the hexadecimal format.

Let's spare ourselves another migration by using the second option:

```rust
//! [...]

async fn validate_credentials(/* */) -> Result<uuid::Uuid, PublishError> {
    let password_hash = sha3::Sha3_256::digest(
        credentials.password.expose_secret().as_bytes()
    );
    // Lowercase hexadecimal encoding.
    let password_hash = format!("{:x}", password_hash);
    // [...]
}
```

The application code should compile now. The test suite, instead, requires a bit more work.
The `test_user` helper method was recovering a set of valid credentials by querying the `users` table - this is no longer viable now that we are storing hashes instead of raw passwords!

```rust
//! tests/api/helpers.rs
//! [...]

impl TestApp {
    // [...]

    pub async fn test_user(&self) -> (String, String) {
        let row = sqlx::query!("SELECT username, password FROM users LIMIT 1",)
            .fetch_one(&self.db_pool)
            .await
            .expect("Failed to create test users.");
        (row.username, row.password)
    }
}

pub async fn spawn_app() -> TestApp {
    // [...]
    let test_app = TestApp {/* */};
    add_test_user(&test_app.db_pool).await;
    test_app
}

async fn add_test_user(pool: &PgPool) {
    sqlx::query!(
        "INSERT INTO users (user_id, username, password)
        VALUES ($1, $2, $3)",
        Uuid::new_v4(),
        Uuid::new_v4().to_string(),
        Uuid::new_v4().to_string(),
    )
    .execute(pool)
    .await
    .expect("Failed to create test users.");
}
```

We need `TestApp` to store the randomly generated password in order for us to access it in our helper methods.
Let's start by creating a new helper struct, `TestUser`:

```rust
//! tests/api/helpers.rs
//! [...]
use sha3::Digest;

pub struct TestUser {
    pub user_id: Uuid,
    pub username: String,
    pub password: String
}

impl TestUser {
    pub fn generate() -> Self {
        Self {
            user_id: Uuid::new_v4(),
            username: Uuid::new_v4().to_string(),
            password: Uuid::new_v4().to_string()
        }
    }

    async fn store(&self, pool: &PgPool) {
        let password_hash = sha3::Sha3_256::digest(
            credentials.password.expose_secret().as_bytes()
        );
        let password_hash = format!("{:x}", password_hash);
        sqlx::query!(
            "INSERT INTO users (user_id, username, password_hash)
            VALUES ($1, $2, $3)",
            self.user_id,
            self.username,
            password_hash,
        )
        .execute(pool)
        .await
        .expect("Failed to store test user.");
    }
}
```

We can then attach an instance of `TestUser` to `TestApp`, as a new field:

```rust
//! tests/api/helpers.rs
//! [...]

pub struct TestApp {
    // [...]
    test_user: TestUser
}

pub async fn spawn_app() -> TestApp {
    // [...]
    let test_app = TestApp {
        // [...]
        test_user: TestUser::generate()
    };
    test_app.test_user.store(&test_app.db_pool).await;
    test_app
}
```

To finish, let's delete `add_test_user`, `TestApp::test_user` and update `TestApp::post_newsletters`:

```rust
//! tests/api/helpers.rs
//! [...]

impl TestApp {
    // [..]
    pub async fn post_newsletters(&self, body: serde_json::Value) -> reqwest::Response {
        reqwest::Client::new()
            .post(&format!("{}/newsletters", &self.address))
            .basic_auth(&self.test_user.username, Some(&self.test_user.password))
            // [...]
    }
}
```

The test suite should now compile and run successfully.

10.2.3.3 Preimage Attack

Is SHA3-256 enough to protect our users' passwords if an attacker gets their hands on our `users` table?

Let's imagine that the attack wants to crack a **specific** password hash in our database.
The attacker does not even need to retrieve the original password. To authenticate successfully they just need to find an input string `s` whose SHA3-256 hash matches the password they are trying to crack - in other words, a collision.
This is known as a preimage attack.

How hard is it?

The math is a bit tricky, but a brute-force attack has an **exponential** time complexity - `2^n`, where `n` is the hash length in bits.
If `n > 128`, it is considered unfeasible to compute.
Unless a vulnerability is found in SHA-3, we do not need to worry about preimage attacks against SHA3-256.

10.2.3.4 Naive Dictionary Attack

We are not hashing arbitrary inputs though - we can reduce the search space by making some assumptions on the original password: how long was it? What symbols were used?

Let's imagine we are looking for an alphanumeric password that is shorter than 17 characters[3].

We can count the number of password candidates:

```
// (26 letters + 10 number symbols) ^ Password Length
// for all allowed password lengths
36^1 +
36^2 +
... +
36^16
```

It sums up to roughly `8 * 10^24` possibilities.
I wasn't able to find data specifically on SHA3-256, but researchers managed to compute ~900 million SHA3-**512** hashes per second using a Graphical Processing Unit (GPU).

Assuming a hash rate of `~10^9` per second, it would take us `~10^15` seconds to hash all password candidates. The approximate age of the universe is `4 * 10^17` seconds.
Even if we were to parallelise our search using a million GPUs, it would still take `~10^9` seconds - roughly 30 years[4].

10.2.3.5 Dictionary Attack

Let's go back to what we discussed at the very beginning of this chapter - it is impossible for a person to remember a unique password for hundreds of online services.
Either they rely on a password manager, or they are re-using one or more passwords across multiple accounts.

Furthermore, most passwords are far from being random, even when reused - common words, full names, dates, names of popular sport teams, etc.
An attacker could easily design a simple algorithm to generate thousands of plausible passwords - but they do not have to. They can look at a password dataset from one of the many security breaches from the last decade to find the most common passwords in the wild.

In a couple of minutes they can pre-compute the SHA3-256 hash of the most commonly used 10 million passwords. Then they start scanning our database looking for a match.

This is known as **dictionary attack** - and it's extremely effective.

All the cryptographic hash functions we mentioned so far are designed to be **fast**.
Fast enough to enable anybody to pull off a dictionary attack without having to use specialised hardware.

We need something **much slower**, but with the same set of mathematical properties of cryptographic hash functions.

[3] When looking into brute-force attacks you will often see mentions of rainbow tables - an efficient data structure to pre-compute and lookup hashes.

[4] This back-of-the-envelope calculation should make it clear that using a randomly-generated password provides you, as a user, with a significant level of protection against brute-force attacks even if the server is using fast hashing algorithms for password storage. Consistent usage of a password manager is indeed one of the easiest ways to boost your security profile.

10.2.3.6 Argon2

The Open Web Application Security Project (OWASP)[5] provides useful guidance on safe password storage - with a whole section on how to choose the correct hashing algorithm:

- Use Argon2id with a minimum configuration of 15 MiB of memory, an iteration count of 2, and 1 degree of parallelism.
- If Argon2id is not available, use bcrypt with a work factor of 10 or more and with a password limit of 72 bytes.
- For legacy systems using scrypt, use a minimum CPU/memory cost parameter of (2^16), a minimum block size of 8 (1024 bytes), and a parallelization parameter of 1.
- If FIPS-140 compliance is required, use PBKDF2 with a work factor of 310,000 or more and set with an internal hash function of HMAC-SHA-256.
- Consider using a pepper to provide additional defense in depth (though alone, it provides no additional secure characteristics).

All these options - Argon2, bcrypt, scrypt, PBKDF2 - are designed to be **computationally demanding**.
They also expose configuration parameters (e.g. work factor for bcrypt) to further slow down hash computation: application developers can tune a few knobs to keep up with hardware speed-ups - no need to migrate to newer algorithms every couple of years.

Let's replace SHA-3 with Argon2id, as recommended by OWASP.
The Rust Crypto organization got us covered once again - they provide a pure-Rust implementation, `argon2`.

Let's add it to our dependencies:

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

[dependencies]
# [...]
argon2 = { version = "0.3", features = ["std"] }
```

To hash a password we need to create an `Argon2` struct instance.
The `new` method signature looks like this:

```rust
//! argon2/lib.rs
/// [...]

impl<'key> Argon2<'key> {
    /// Create a new Argon2 context.
    pub fn new(algorithm: Algorithm, version: Version, params: Params) -> Self {
        // [...]
    }
```

[5] OWASP is, generally speaking, a treasure trove of great educational material about security for web applications. You should get as familiar as possible with OWASP's material, especially if you do not have an application security specialist in your team/organization to support you. On top of the cheatsheet we linked, make sure to browse their Application Security Verification Standard.

```
    // [...]
}
```

`Algorithm` is an enum: it lets us select which variant of Argon2 we want to use - Argon2d, Argon2i, Argon2id. To comply with OWASP's recommendation we will go for `Algorithm::Argon2id`.

`Version` fulfills a similar purpose - we will go for the most recent, `Version::V0x13`.

What about `Params`?
`Params::new` specifies all the mandatory parameters we need to provide to build one:

```
//! argon2/params.rs
// [...]

/// Create new parameters.
pub fn new(
    m_cost: u32,
    t_cost: u32,
    p_cost: u32,
    output_len: Option<usize>
) -> Result<Self> {
    // [...]
}
```

`m_cost`, `t_cost` and `p_cost` map to OWASP's requirements:

- `m_cost` is the memory size, expressed in kilobytes
- `t_cost` is the number of iterations;
- `p_cost` is the degree of parallelism.

`output_len`, instead, determines the length of the returned hash - if omitted, it will default to 32 bytes. That is equal to 256 bits, the same hash length we were getting via SHA3-256.

We know enough, at this point, to build one:

```
//! src/routes/newsletters.rs
use argon2::{Algorithm, Argon2, Version, Params};
// [...]

async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let hasher = Argon2::new(
        Algorithm::Argon2id,
        Version::V0x13,
        Params::new(15000, 2, 1, None)
            .context("Failed to build Argon2 parameters")
            .map_err(PublishError::UnexpectedError)?,
    );
    let password_hash = sha3::Sha3_256::digest(
        credentials.password.expose_secret().as_bytes()
    );
    // [...]
}
```

`Argon2` implements the `PasswordHasher` trait:

```rust
//! password_hash/traits.rs

pub trait PasswordHasher {
    // [...]
    fn hash_password<'a, S>(
        &self,
        password: &[u8],
        salt: &'a S
    ) -> Result<PasswordHash<'a>>
    where
        S: AsRef<str> + ?Sized;
}
```

It is a re-export from the `password-hash` crate, a unified interface to work with password hashes backed by a variety of algorithm (currently Argon2, PBKDF2 and scrypt).

`PasswordHasher::hash_password` is a bit different from `Sha3_256::digest` - it is asking for an additional parameter on top of the raw password, a `salt`.

10.2.3.7 Salting

Argon2 is a lot slower than SHA-3, but this is not enough to make a dictionary attack unfeasible. It takes longer to hash the most common 10 million passwords, but not prohibitively long.

What if, though, the attacker had to rehash the whole dictionary **for every user in our database**? It becomes a lot more challenging!

That is what **salting** accomplishes. For each user, we generate a **unique** random string - the salt. The salt is prepended to the user password before generating the hash. `PasswordHasher::hash_password` takes care of the prepending business for us.

The salt is stored next to the password hash, in our database.
If an attacker gets their hands on a database backup, they will have access to all salts[6].
But they have to compute `dictionary_size * n_users` hashes instead of `dictionary_size`.
Furthermore, pre-computing the hashes is no longer an option - this buys us time to detect the breach and take action (e.g. force a password reset for all users).

Let's add a `password_salt` column to our `users` table:

```
sqlx migrate add add_salt_to_users
```

```sql
-- migrations/20210815112111_add_salt_to_users.sql
ALTER TABLE users ADD COLUMN salt TEXT NOT NULL;
```

We can no longer compute the hash before querying the `users` table - we need to retrieve the salt first.
Let's shuffle operations around:

[6] This is why OWASP recommends an additional layer of defence - **peppering**. All hashes stored in the database are encrypted using a shared secret, only known to the application. Encryption, though, brings its own set of challenges: where are we going to store the key? How do we rotate it? The answer usually involves a Hardware Security Module (HSM) or a secret vault, such as AWS CloudHSM, AWS KMS or Hashicorp Vault. A thorough overview of key management is beyond the scope of this book.

```rust
//! src/routes/newsletters.rs
// [...]
use argon2::PasswordHasher;

async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let hasher = argon2::Argon2::new(/* */);
    let row: Option<_> = sqlx::query!(
        r#"
        SELECT user_id, password_hash, salt
        FROM users
        WHERE username = $1
        "#,
        credentials.username,
    )
    .fetch_optional(pool)
    .await
    .context("Failed to perform a query to retrieve stored credentials.")
    .map_err(PublishError::UnexpectedError)?;

    let (expected_password_hash, user_id, salt) = match row {
        Some(row) => (row.password_hash, row.user_id, row.salt),
        None => {
            return Err(PublishError::AuthError(anyhow::anyhow!(
                "Unknown username."
            )));
        }
    };

    let password_hash = hasher
        .hash_password(
            credentials.password.expose_secret().as_bytes(),
            &salt
        )
        .context("Failed to hash password")
        .map_err(PublishError::UnexpectedError)?;

    let password_hash = format!("{:x}", password_hash.hash.unwrap());

    if password_hash != expected_password_hash {
        Err(PublishError::AuthError(anyhow::anyhow!(
            "Invalid password."
        )))
    } else {
        Ok(user_id)
    }
}
```

Unfortunately, this does not compile:

```
error[E0277]: the trait bound
`argon2::password_hash::Output: LowerHex` is not satisfied
   --> src/routes/newsletters.rs
    |
125 |     let password_hash = format!("{:x}", password_hash.hash.unwrap());
```

```
    |                                          ^^^^^^^^^^^^^^^^^^^^^^^^^^^
    the trait `LowerHex` is not implemented for `argon2::password_hash::Output`
```

`Output` provides other methods to obtain a string representation - e.g. `Output::b64_encode`. It would work, as long as we are happy to change the assumed encoding for hashes stored in our database.

Given that a change is necessary, we can shoot for something better than base64-encoding.

10.2.3.8 PHC String Format

To authenticate a user, we need **reproducibility**: we must run the very same hashing routine every single time.
Salt and password are just *a subset* of the inputs for Argon2id. All the other load parameters (`t_cost`, `m_cost`, `p_cost`) are equally important to obtain the same hash given the same pair of salt and password.

If we store a base64-encoded representation of the hash, we are making a strong implicit assumption: all values stored in the `password_hash` column have been computed using the same load parameters.

As we discussed a few sections ago, hardware capabilities evolve over time: application developers are expected to keep up by increasing the computational cost of hashing using higher load parameters.
What happens when you have to migrate your stored passwords to a newer hashing configuration?

To keep authenticating old users we **must** store, next to each hash, the exact set of load parameters used to compute it.
This allows for a seamless migration between two different load configurations: when an old user authenticates, we verify password validity using the stored load parameters; we then recompute the password hash using the new load parameters and update the stored information accordingly.

We could go for the naive approach - add three new columns to our `users` table: `t_cost`, `m_cost` and `p_cost`.
It would work, as long as the algorithm remains Argon2id.

What happens if a vulnerability is found in Argon2id and we are forced to migrate away from it? We'd probably want to add an `algorithm` column, as well as new columns to store the load parameters of Argon2id's replacement.

It can be done, but it is tedious.
Luckily enough, there is a better solution: the PHC string format. The PHC string format provides a standard representation for a password hash: it includes the hash itself, the salt, the algorithm and all its associated parameters.

Using the PHC string format, an Argon2id password hash looks like this:

```
# ${algorithm}${algorithm version}${$-separated algorithm parameters}${hash}${salt}
$argon2id$v=19$m=65536,t=2,p=1$gZiV/M1gPc22ElAH/Jh1Hw$CWOrkoo7oJBQ/iyh7uJ0LO2aLEfrHwTWllSAxT0zRno
```

The `argon2` crate exposes `PasswordHash`, a Rust implementation of the PHC format:

```rust
//! argon2/lib.rs
// [...]

pub struct PasswordHash<'a> {
    pub algorithm: Ident<'a>,
    pub version: Option<Decimal>,
    pub params: ParamsString,
    pub salt: Option<Salt<'a>>,
    pub hash: Option<Output>,
}
```

Storing password hashes in PHC string format spares us from having to initialise the `Argon2` struct using explicit parameters[7].
We can rely on `Argon2`'s implementation of the `PasswordVerifier` trait:

```rust
pub trait PasswordVerifier {
    fn verify_password(
        &self,
        password: &[u8],
        hash: &PasswordHash<'_>
    ) -> Result<()>;
}
```

By passing the expected hash via `PasswordHash`, `Argon2` can automatically infer what load parameters and salt should be used to verify if the password candidate is a match[8].

Let's update our implementation:

```rust
//! src/routes/newsletters.rs
use argon2::{Argon2, PasswordHash, PasswordVerifier};
// [...]


async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let row: Option<_> = sqlx::query!(
        r#"
        SELECT user_id, password_hash
        FROM users
        WHERE username = $1
        "#,
        credentials.username,
    )
    .fetch_optional(pool)
```

[7] I have not delved too deep into the source code of the different hash algorithms that implement `PasswordVerifier`, but I do wonder why `verify_password` needs to take `&self` as a parameter. `Argon2` has absolutely no use for it, but it forces us to go through an `Argon2::default` in order to call `verify_password`.

[8] `PasswordVerifier::verify_password` does one more thing - it leans on `Output` to compare the two hashes, instead of working with raw bytes. `Output`'s implementations of `PartialEq` and `Eq` are designed to be evaluated in constant-time - no matter how different or similar the inputs are, function execution will take the same amount of time. Assuming an attacker had perfect knowledge of the hashing algorithm configuration the server is using, they could analyze the response time for each authentication attempt to infer the first bytes of the password hash - combined with a dictionary, this could help them to crack the password. The feasibility of such an attack is debatable, even more so when salting is in place. Nonetheless, it costs us nothing - so better safe than sorry.

```rust
    .await
    .context("Failed to perform a query to retrieve stored credentials.")
    .map_err(PublishError::UnexpectedError)?;

    let (expected_password_hash, user_id) = match row {
        Some(row) => (row.password_hash, row.user_id),
        None => {
            return Err(PublishError::AuthError(anyhow::anyhow!(
                "Unknown username."
            )))
        }
    };

    let expected_password_hash = PasswordHash::new(&expected_password_hash)
        .context("Failed to parse hash in PHC string format.")
        .map_err(PublishError::UnexpectedError)?;

    Argon2::default()
        .verify_password(
            credentials.password.expose_secret().as_bytes(),
            &expected_password_hash
        )
        .context("Invalid password.")
        .map_err(PublishError::AuthError)?;

    Ok(user_id)
}
```

It compiles successfully.
You might have also noticed that we no longer deal with the salt directly - PHC string format takes care of it for us, implicitly.
We can get rid of the `salt` column entirely:

```
sqlx migrate add remove_salt_from_users
```

```sql
-- migrations/20210815112222_remove_salt_from_users.sql
ALTER TABLE users DROP COLUMN salt;
```

What about our tests?
Two of them are failing:

```
---- newsletter::newsletters_are_not_delivered_to_unconfirmed_subscribers stdout ----
'newsletter::newsletters_are_not_delivered_to_unconfirmed_subscribers' panicked at
'assertion failed: `(left == right)`
  left: `500`,
 right: `200`',

---- newsletter::newsletters_are_delivered_to_confirmed_subscribers stdout ----
'newsletter::newsletters_are_delivered_to_confirmed_subscribers' panicked at
'assertion failed: `(left == right)`
  left: `500`,
```

We can look at logs to figure out what is wrong:

```
TEST_LOG=true cargo t newsletters_are_not_delivered | bunyan
```

```
[2021-08-29T20:14:50.367Z] ERROR: [HTTP REQUEST - EVENT]
  Error encountered while processing the incoming HTTP request:
  Failed to parse hash in PHC string format.

  Caused by:
     password hash string invalid
```

Let's look at the password generation code for our test user:

```rust
//! tests/api/helpers.rs
// [ ... ]

impl TestUser {
    // [ ... ]
    async fn store(&self, pool: &PgPool) {
        let password_hash = sha3::Sha3_256::digest(
            credentials.password.expose_secret().as_bytes()
        );
        let password_hash = format!("{:x}", password_hash);
        // [ ... ]
    }
}
```

We are still using SHA-3!
Let's update it:

```rust
//! tests/api/helpers.rs
use argon2::password_hash::SaltString;
use argon2::{Argon2, PasswordHasher};
// [ ... ]

impl TestUser {
    // [ ... ]
    async fn store(&self, pool: &PgPool) {
        let salt = SaltString::generate(&mut rand::thread_rng());
        // We don't care about the exact Argon2 parameters here
        // given that it's for testing purposes!
        let password_hash = Argon2::default()
            .hash_password(self.password.as_bytes(), &salt)
            .unwrap()
            .to_string();
        // [ ... ]
    }
}
```

The test suite should pass now.
We have removed all mentions of `sha3` from our project - we can now remove it from the list of dependencies in `Cargo.toml`.

10.2.4 Do Not Block The Async Executor

How long is it taking to verify user credentials when running our integration tests?
We currently do not have a span around password hashing - let's fix it:

```rust
//! src/routes/newsletters.rs
// [ ... ]
```

```rust
#[tracing::instrument(name = "Validate credentials", skip(credentials, pool))]
async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let (user_id, expected_password_hash) = get_stored_credentials(
            &credentials.username,
            &pool
        )
        .await
        .map_err(PublishError::UnexpectedError)?
        .ok_or_else(|| PublishError::AuthError(anyhow::anyhow!("Unknown username.")))?;

    let expected_password_hash = PasswordHash::new(
            &expected_password_hash.expose_secret()
        )
        .map_err(PublishError::UnexpectedError)?;

    tracing::info_span!("Verify password hash")
        .in_scope(|| {
            Argon2::default()
                .verify_password(
                    credentials.password.expose_secret().as_bytes(),
                    &expected_password_hash
                )
        })
        .context("Invalid password.")
        .map_err(PublishError::AuthError)?;

    Ok(user_id)
}

// We extracted the db-querying logic in its own function with its own span.
#[tracing::instrument(name = "Get stored credentials", skip(username, pool))]
async fn get_stored_credentials(
    username: &str,
    pool: &PgPool,
) -> Result<Option<(uuid::Uuid, Secret<String>)>, anyhow::Error> {
    let row = sqlx::query!(
        r#"
        SELECT user_id, password_hash
        FROM users
        WHERE username = $1
        "#,
        username,
    )
    .fetch_optional(pool)
    .await
    .context("Failed to perform a query to retrieve stored credentials.")?
    .map(|row| (row.user_id, Secret::new(row.password_hash)));
    Ok(row)
}
```

We can now look at the logs from one of our integration tests:

```
TEST_LOG=true cargo test --quiet --release \
  newsletters_are_delivered | grep "VERIFY PASSWORD" | bunyan
```

```
[ ... ]  [VERIFY PASSWORD HASH - END] (elapsed_milliseconds=11, ... )
```

Roughly 10ms.
This is likely to cause issues under load - the infamous **blocking** problem.

`async/await` in Rust is built around a concept called **cooperative scheduling**.

How does it work?
Let's look at an example:

```
async fn my_fn() {
    a().await;
    b().await;
    c().await;
}
```

`my_fn` returns a `Future`.
When the future is awaited, our async runtime (`tokio`) enters into the picture: it starts `poll`ing it.

How is `poll` implemented for the `Future` returned by `my_fn`?
You can think of it as a state machine:

```
enum MyFnFuture {
    Initialized,
    CallingA,
    CallingB,
    CallingC,
    Complete
}
```

Every time `poll` is called, it tries to make progress by reaching the next state. E.g. if `a.await()` has returned, we start awaiting `b()`[9].

We have a different state in `MyFnFuture` for each `.await` in our async function body.
This is why `.await` calls are often named **yield points** - our future progresses from the previous `.await` to the next one and then **yields** control back to the executor.

The executor can then choose to poll the same future again or to prioritise making progress on another task. This is how async runtimes, like `tokio`, manage to make progress **concurrently** on multiple tasks - by continuously parking and resuming each of them.
In a way, you can think of async runtimes as great jugglers.

The underlying assumption is that most async tasks are performing some kind of input-output (IO) work - most of their execution time will be spent waiting on something else to happen (e.g. the operating system notifying us that there is data ready to be read on a socket), therefore we can *effectively* perform many more tasks concurrently than we what we would achieve by dedicating a parallel unit of execution (e.g. one thread per OS core) to each task.

This model works great assuming tasks **cooperate** by frequently yielding control back to the executor.

[9] Our example is oversimplified, on purpose. In reality, each of those states will have sub-states in turn - one for each `.await` in the body of the function we are calling. A future can turn into a deeply nested state machine!

In other words, `poll` is expected to be **fast** - it should return in less than 10-100 microseconds[10]. If a call to `poll` takes longer (or, even worse, never returns), then the async executor cannot make progress on any other task - this is what people refer to when they say that "a task is blocking the executor/the async thread".

You should always be on the lookout for CPU-intensive workloads that are likely to take longer than 1ms - password hashing is a perfect example.
To play nicely with `tokio`, we must offload our CPU-intensive task to a separate threadpool using `tokio::task::spawn_blocking`. Those threads are reserved for blocking operations and do not interfere with the scheduling of async tasks.

Let's get to work!

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(name = "Validate credentials", skip(credentials, pool))]
async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    // [...]
    tokio::task::spawn_blocking(move || {
        tracing::info_span!("Verify password hash").in_scope(|| {
            Argon2::default()
                .verify_password(
                    credentials.password.expose_secret().as_bytes(),
                    expected_password_hash
                )
        })
    })
    .await
    // spawn_blocking is fallible - we have a nested Result here!
    .context("Failed to spawn blocking task.")
    .map_err(PublishError::UnexpectedError)?
    .context("Invalid password.")
    .map_err(PublishError::AuthError)?;
    // [...]
}
```

The borrow checker is not happy:

```
error[E0597]: `expected_password_hash` does not live long enough
   --> src/routes/newsletters.rs
    |
117 |     PasswordHash::new(&expected_password_hash)
    |     ------------------^^^^^^^^^^^^^^^^^^^^^^^_
    |     |                 |
    |     |                 borrowed value does not live long enough
    |     argument requires that `expected_password_hash` is borrowed for `'static`
...
134 | }
    | - `expected_password_hash` dropped here while still borrowed
```

[10] This heuristic is reported in "Async: What is blocking?" by Alice Rhyl, one of `tokio`'s maintainers. An article I'd strongly suggest you to read to understand better the underlying mechanics of `tokio` and `async/await` in general!

We are launching a computation on a separate thread - the thread itself might outlive the async task we are spawning it from. To avoid the issue, `spawn_blocking` requires its argument to have a `'static` lifetime - which is preventing us from passing references to the current function context into the closure.

You might argue - "We are using `move || {}`, the closure should be taking ownership of `expected_password_hash`!".
You would be right! But that is not enough.
Let's look again at how `PasswordHash` is defined:

```rust
pub struct PasswordHash<'a> {
    pub algorithm: Ident<'a>,
    pub salt: Option<Salt<'a>>,
    // [...]
}
```

It holds a reference to the string it was parsed from.
We need to move ownership of the original string into our closure, moving the parsing logic into it as well.

Let's create a separate function, `verify_password_hash`, for clarity:

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(name = "Validate credentials", skip(credentials, pool))]
async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    // [...]
    tokio::task::spawn_blocking(move || {
        verify_password_hash(
            expected_password_hash,
            credentials.password
        )
    })
    .await
    .context("Failed to spawn blocking task.")
    .map_err(PublishError::UnexpectedError)??;

    Ok(user_id)
}

#[tracing::instrument(
    name = "Verify password hash",
    skip(expected_password_hash, password_candidate)
)]
fn verify_password_hash(
    expected_password_hash: Secret<String>,
    password_candidate: Secret<String>,
) -> Result<(), PublishError> {
    let expected_password_hash = PasswordHash::new(
            expected_password_hash.expose_secret()
        )
        .context("Failed to parse hash in PHC string format.")
        .map_err(PublishError::UnexpectedError)?;

    Argon2::default()
        .verify_password(
             password_candidate.expose_secret().as_bytes(),
             &expected_password_hash
        )
        .context("Invalid password.")
        .map_err(PublishError::AuthError)
}
```

It compiles!

10.2.4.1 Tracing Context Is Thread-Local

Let's look again at the logs for the `verify password hash` span:

```
TEST_LOG=true cargo test --quiet --release \
  newsletters_are_delivered | grep "VERIFY PASSWORD" | bunyan
```

```
[2021-08-30T10:03:07.613Z]  [VERIFY PASSWORD HASH - START]
  (file="...", line="...", target="...")
[2021-08-30T10:03:07.624Z]  [VERIFY PASSWORD HASH - END]
  (file="...", line="...", target="...")
```

We are missing all the properties that are inherited from the root span of the corresponding request - e.g. `request_id`, `http.method`, `http.route`, etc. Why?

Let's look at `tracing`'s documentation:

> Spans form a tree structure — unless it is a root span, all spans have a *parent*, and may have one or more *children*. When a new span is created, the **current span** becomes the new span's parent.

The current span is the one returned by `tracing::Span::current()` - let's check its documentation:

> Returns a handle to the span considered by the `Collector` to be the current span.
> If the collector indicates that it does not track the current span, or that **the thread from which this function is called is not currently inside a span**, the returned span will be disabled.

"Current span" actually means "active span for the current thread".
That is why we are not inheriting any property: we are spawning our computation on a separate thread and `tracing::info_span!` does not find any active `Span` associated with it when it executes.

We can work around the issue by explicitly attaching the current span to the newly spawn thread:

```
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(name = "Validate credentials", skip(credentials, pool))]
```

```rust
async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    // [...]
    // This executes before spawning the new thread
    let current_span = tracing::Span::current();
    tokio::task::spawn_blocking(move || {
        // We then pass ownership to it into the closure
        // and explicitly executes all our computation
        // within its scope.
        current_span.in_scope(|| {
            verify_password_hash(/* */)
        })
    })
    // [...]
}
```

You can verify that it works - we are now getting all the properties we care about.
It is a bit verbose though - let's write a helper function:

```rust
//! src/telemetry.rs
use tokio::task::JoinHandle;
// [...]

// Just copied trait bounds and signature from `spawn_blocking`
pub fn spawn_blocking_with_tracing<F, R>(f: F) -> JoinHandle<R>
where
    F: FnOnce() -> R + Send + 'static,
    R: Send + 'static,
{
    let current_span = tracing::Span::current();
    tokio::task::spawn_blocking(move || current_span.in_scope(f))
}
```

```rust
//! src/routes/newsletters.rs
use crate::telemetry::spawn_blocking_with_tracing;
// [...]

#[tracing::instrument(name = "Validate credentials", skip(credentials, pool))]
async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    // [...]
    spawn_blocking_with_tracing(move || {
        verify_password_hash(/* */)
    })
    // [...]
}
```

We can now easily reach for it every time we need to offload some CPU-intensive computation to a dedicated threadpool.

10.2.5 User Enumeration

Let's add a new test case:

```rust
//! tests/api/newsletter.rs
use uuid::Uuid;
// [...]

#[tokio::test]
async fn non_existing_user_is_rejected() {
    // Arrange
    let app = spawn_app().await;
    // Random credentials
    let username = Uuid::new_v4().to_string();
    let password = Uuid::new_v4().to_string();

    let response = reqwest::Client::new()
        .post(&format!("{}/newsletters", &app.address))
        .basic_auth(username, Some(password))
        .json(&serde_json::json!({
            "title": "Newsletter title",
            "content": {
                "text": "Newsletter body as plain text",
                "html": "<p>Newsletter body as HTML</p>",
            }
        }))
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(401, response.status().as_u16());
    assert_eq!(
        r#"Basic realm="publish""#,
        response.headers()["WWW-Authenticate"]
    );
}
```

The test should pass straight-away.
How long does it take though?

Let's look at the logs!

```bash
TEST_LOG=true cargo test --quiet --release \
  non_existing_user_is_rejected | grep "HTTP REQUEST" | bunyan
```

```
# [...] Omitting setup requests
[...] [HTTP REQUEST - END]
  (http.route = "/newsletters", elapsed_milliseconds=1, ...)
```

Roughly 1ms.

Let's add another test: this time we pass a valid username with an incorrect password.

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
```

```
async fn invalid_password_is_rejected() {
    // Arrange
    let app = spawn_app().await;
    let username = &app.test_user.username;
    // Random password
    let password = Uuid::new_v4().to_string();
    assert_ne!(app.test_user.password, password);

    let response = reqwest::Client::new()
        .post(&format!("{}/newsletters", &app.address))
        .basic_auth(username, Some(password))
        .json(&serde_json::json!({
            "title": "Newsletter title",
            "content": {
                "text": "Newsletter body as plain text",
                "html": "<p>Newsletter body as HTML</p>",
            }
        }))
        .send()
        .await
        .expect("Failed to execute request.");

    // Assert
    assert_eq!(401, response.status().as_u16());
    assert_eq!(
        r#"Basic realm="publish""#,
        response.headers()["WWW-Authenticate"]
    );
}
```

This one should pass as well. How long does the request take to fail?

```
TEST_LOG=true cargo test --quiet --release \
  invalid_password_is_rejected | grep "HTTP REQUEST" | bunyan
```

```
# [...] Omitting setup requests
[...] [HTTP REQUEST - END]
  (http.route = "/newsletters", elapsed_milliseconds=11, ...)
```

Roughly 10ms - it is one order of magnitude smaller!
We can use this difference to our advantage to perform a **timing attack**, a member of the broader class of side-channel attacks.

If an attacker knows at least one valid username, they can inspect the server response times[11] to confirm if another username exists or not - we are looking at a potential user enumeration vulnerability.
Is this an issue?

It depends.
If you are running Gmail, there are plenty of other ways to find out if a `@gmail.com` email address exists or not. The validity of an email address is not a secret!

[11] In a real life scenario, there is a network between an attacker and your server. Load and network variance are likely to mask the speed difference on a limited set of attempts, but if you collect enough data points it should be possible to notice a statistically significant difference in latency.

If you are running a SaaS product, the situation might be more nuanced.
Let's go for a fictional scenario: your SaaS product provides payroll services and uses email addresses as usernames. There are separate employee and admin login pages.
My goal is to get access to payroll data - I need to compromise an employee with privileged access.
We can scrape LinkedIn to get the name and surnames of all employees in the Finance department.
Corporate emails follow a predictable structure (name.surname@payrollaces.com), so we have a list of candidates.
We can now perform a timing attack against the admin login page to narrow down the list to those who have access.

Even in our fictional example, user enumeration is not enough, on its own, to escalate our privileges.
But it can be used as a stepping stone to narrow down a set of targets for a more precise attack.

How do we prevent it?
Two strategies:

1. Remove the timing difference between an auth failure due to an invalid password and an auth failure due to a non-existent username;
2. Limit the number of failed auth attempts for a given IP/username.

The second is generally valuable as a protection against brute-force attacks, but it requires holding some state - we will leave it for later.

Let's focus on the first one.
To eliminate the timing difference, we need to perform the **same amount of work** in both cases.

Right now, we follow this recipe:

- Fetch stored credentials for given username;
- If they do not exist, return 401;
- If they exist, hash the password candidate and compare with the stored hash.

We need to remove that early exit - we should have a fallback expected password (with salt and load parameters) that can be compared to the hash of the password candidate.

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(name = "Validate credentials", skip(credentials, pool))]
async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
) -> Result<uuid::Uuid, PublishError> {
    let mut user_id = None;
    let mut expected_password_hash = Secret::new(
        "$argon2id$v=19$m=15000,t=2,p=1$\
        gZiV/M1gPc22ElAH/Jh1Hw$\
        CWOrkoo7oJBQ/iyh7uJ0LO2aLEfrHwTWllSAxT0zRno"
            .to_string()
    );

    if let Some((stored_user_id, stored_password_hash)) =
        get_stored_credentials(&credentials.username, &pool)
            .await
            .map_err(PublishError::UnexpectedError)?
    {
        user_id = Some(stored_user_id);
        expected_password_hash = stored_password_hash;
    }

    spawn_blocking_with_tracing(move || {
        verify_password_hash(expected_password_hash, credentials.password)
    })
    .await
    .context("Failed to spawn blocking task.")
    .map_err(PublishError::UnexpectedError)??;

    // This is only set to `Some` if we found credentials in the store
    // So, even if the default password ends up matching (somehow)
    // with the provided password,
    // we never authenticate a non-existing user.
    // You can easily add a unit test for that precise scenario.
    user_id.ok_or_else(||
        PublishError::AuthError(anyhow::anyhow!("Unknown username."))
    )
}
```

```rust
//! tests/api/helpers.rs
use argon2::{Algorithm, Argon2, Params, PasswordHasher, Version};
// [...]

impl TestUser {
    async fn store(&self, pool: &PgPool) {
        let salt = SaltString::generate(&mut rand::thread_rng());
        // Match parameters of the default password
        let password_hash = Argon2::new(
            Algorithm::Argon2id,
            Version::V0x13,
            Params::new(15000, 2, 1, None).unwrap(),
        )
        .hash_password(self.password.as_bytes(), &salt)
        .unwrap()
        .to_string();
        // [...]
    }
    // [...]
}
```

There should not be any statistically significant timing difference now.

10.3 Is it safe?

We went to great lengths to follow all most common best practices while building our password-based authentication flow.
Time to ask ourselves: is it safe?

10.3.1 Transport Layer Security (TLS)

We are using the 'Basic' Authentication Scheme to pass credentials between the client and the server - username and password are encoded, but not encrypted.
We **must** use Transport Layer Security (TLS) to ensure that nobody can eavesdrop the traffic between client and server to compromise the user credentials (a man-in-the-middle attack - MITM)[12].
Our API is already served via HTTPS, so nothing to do here.

10.3.2 Password Reset

What happens if an attacker manages to steal a set of valid user credentials?
Passwords do not expire - they are long-lived secrets.

Right now, there is no way for a user to reset their passwords. This is definitely a gap we'd need to fill.

10.3.3 Interaction Types

So far we have been fairly vague about **who** is calling to our API.

The **type of interaction** we need to support is a key decision factor when it comes to authentication.

We will look at three categories of callers:

- Other APIs (machine-to-machine);
- A person, via a browser;
- Another API, on behalf of a person.

10.3.4 Machine To Machine

The consumer of your API might be a machine (e.g. another API).
This is often the case in a microservice architecture - your functionality emerges from a variety of services interacting over the network.

To significantly raise our security profile we'd have to throw in *something they have* (e.g. request signing) or *something they are* (e.g. IP range restrictions).
A popular option, when all service are owned by the same organization, is mutual TLS (mTLS).

Both signing and mTLS rely on of public key cryptography - keys must be provisioned, rotated, managed. The overhead is only justified once your system reaches a certain size.

10.3.4.1 Client Credentials via OAuth2

Another option is using the OAuth2 client credentials flow.

APIs no longer have to manage passwords (client secrets, in OAuth2 terminology) - the concern is delegated to a centralised authorization server. There are multiple turn-key implementations of an authorization server out there - both OSS and commercial. You can lean on them instead of rolling your own.

[12] Which is why you should never enter your password into a website that is not using HTTPS - i.e. HTTP + TLS.

The caller authenticates with the authorization server - if successful, the auth server grants them a set of temporary credentials (a JWT access token) which can be used to call our API.
Our API can verify the validity of the access token using public key cryptography, without having to keep any state. **Our API never sees the actual password**, the client secret.

JWT validation is not without its risks - the specification is riddled with dangerous edge cases.

10.3.5 Person Via Browser

What if we are dealing with a person, using a web browser?

'Basic' Authentication requires the client to present their credentials on **every single request**.
We now have a single protected endpoint, but you can easily picture a situation with five or ten pages providing privileged functionality. As it stands, 'Basic' Authentication would force the user to submit their credentials on **every single page**. Not great.

We need a way to *remember* that a user authenticated a few moments ago - i.e. to attach some kind of state to a sequence of requests coming from the same browser. This is accomplished using **sessions**.

A user is asked to authenticate once, via a login form[13]: if successful, the server generates a one-time secret - an authenticated session token. The token is stored in the browser as a secure cookie.
Sessions, unlike passwords, are designed to expire - this reduces the likelihood that a valid session token is compromised (especially if inactive users are automatically logged out). It also prevents the user from having to reset their password if there is a suspicion that their session has been hijacked - a forced log out is much more acceptable than an automated password reset.

This approach is often referred to as **session-based authentication**.

10.3.5.1 Federated Identity

With session-based authentication we still have an authentication step to take care of - the login form.
We can keep rolling our own - everything we learned about passwords is still relevant, even if we ditch the 'Basic' Authentication scheme.

Many websites choose to offer their users an additional option: login via a Social profile - e.g. "Log in with Google". This removes friction from the sign-up flow (no need to create yet another password!), increasing conversion - a desirable outcome.

Social logins rely on **identity federation** - we **delegate** the authentication step to a third-party **identity provider**, which in turn shares with us the pieces of information we asked for (e.g. email address, full name and date of birth).

A common implementation of identity federation relies on OpenID Connect, an identity layer on top of the OAuth2 standard.

[13] You could actually use 'Basic' authentication to perform the login step, while relying on session-based authentication for later interactions.

10.3.6 Machine to machine, on behalf of a person

There is one more scenario: a person authorising a machine (e.g. a third-party service) to perform actions against our API on their behalf.
E.g. a mobile app that provides an alternative UI for Twitter.

It is important to stress how this differs from the first scenario we reviewed, pure machine-to-machine authentication.
In this case, the third-party service is not authorised, **on its own**, to perform any action against our API. The third-party service can only perform actions against our API if a user grants them access, **scoped to their set of permissions**.
I can install a mobile app to write tweets on my behalf, but I can't authorise it to tweet on behalf of David Guetta.

'Basic' authentication would be a very poor fit here: we do not want to share our password with a third-party app. The more parties get to see our password, the more likely it is to be compromised.

Furthermore, keeping an audit trail with shared credentials is a nightmare. When something goes wrong, it is impossible to determine **who** did what: was it actually me? Was it one of the twenty apps I shared credentials with? Who takes responsibility?

This is the textbook scenario for OAuth2 - the third-party never gets to see our username and password. They receive an opaque access token from the authentication server which our API knows how to inspect to grant (or deny) access.

10.4 Interlude: Next Steps

Browsers are our main target - it's decided. Our authentication strategy needs to evolve accordingly!

We will first convert our 'Basic' Authentication flow into a login form with session-based auth.
We will build - from scratch - an admin dashboard. It will include a login form, a logout link and a form to change your password. It will give us an opportunity to discuss a few security challenges (i.e. XSS), introduce new concepts (e.g. cookies, HMAC tags) and try out new tooling (e.g. flash messages, `actix-session`).

Let's get to work!

10.5 Login Forms

10.5.1 Serving HTML Pages

So far we have steered away from the complexity of browsers and web pages - it helped us in limiting the number of new concepts we had to pick up early on in our learning journey.
We have now built enough expertise to make the jump - we will handle both the HTML page and the payload submission for our login form.

Let's start from the basics: how do we return an HTML page from our API?
We can begin by adding a dummy home page endpoint.

```rust
//! src/routes/mod.rs

// [...]
// New module!
mod home;
pub use home::*;
```

```rust
//! src/routes/home/mod.rs
use actix_web::HttpResponse;

pub async fn home() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

```rust
//! src/startup.rs
use crate::routes::home;
// [...]

fn run(/* */) -> Result</* */> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/", web::get().to(home))
            // [...]
    })
    // [...]
}
```

Not much to be seen here - we are just returning a `200 OK` without a body.
Let's add a very simple HTML landing page[14] to the mix:

```html
<!-- src/routes/home/home.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Home</title>
    </head>
    <body>
        <p>Welcome to our newsletter!</p>
    </body>
</html>
```

We want to read this file and return it as the body of our `GET /` endpoint.
We can use `include_str!`, a macro from Rust's standard library: it reads the file at the provided path and returns its content as a `&'static str`.
This is possible because `include_str!` operates **at compile-time** - the file content is stored as part of the application binary, therefore ensuring that a pointer to its content (`&str`) remains valid indefinitely (`'static`)[15].

[14] An in-depth introduction to HTML and CSS is beyond the scope of this book. We will avoid CSS entirely while explaining the required basics of HTML as we introduce new elements to build the pages we need for our newsletter application. Check out `Interneting is hard (but it doesn't have to be)` for an excellent introduction to these topics.

[15] There is often confusion around `'static` due to its different meanings depending on the context. Check out this excellent piece on common Rust lifetime misconceptions if you want to learn more about the topic.

```rust
//! src/routes/home/mod.rs
// [...]

pub async fn home() -> HttpResponse {
    HttpResponse::Ok().body(include_str!("home.html"))
}
```

If you launch your application with `cargo run` and visit `http://localhost:8000` in the browser you should see the `Welcome to our newsletter!` message.
The browser is not entirely happy though - if you open the browser's console[16], you should see a warning.
On Firefox 93.0:

> The character encoding of the HTML document was not declared.
> The document will render with garbled text in some browser configurations if the document contains characters from outside the US-ASCII range.
> The character encoding of the page must be declared in the document or in the transfer protocol.

In other words - the browser has *inferred* that we are returning HTML content, but it would very much prefer to be told explicitly.
We have two options:

- Add a special HTML `meta` tag in the document;
- Set the `Content-Type` HTTP header ("transfer protocol").

Better to use both.
Embedding the information inside the document works nicely for browsers and bot crawlers (e.g. Googlebot) while the `Content-Type` HTTP header is understood by all HTTP clients, not just browsers.

When returning an HTML page, the content type should be set to `text/html; charset=utf-8`.
Let's add it in:

```html
<!-- src/routes/home/home.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
        <!-- This is equivalent to a HTTP header -->
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Home</title>
    </head>
    <!-- [...] -->
</html>
```

```rust
//! src/routes/home/mod.rs
// [...]
use actix_web::http::header::ContentType;

pub async fn home() -> HttpResponse {
```

[16] Throughout this chapter we will rely on the introspection tools made available by browsers. For Firefox, follow this guide. For Google Chrome, follow this guide.

```rust
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(include_str!("home.html"))
}
```

The warning should have disappeared from your browsers' console.

Congrats, you have just served your first well-formed web page!

10.6 Login

Let's start working on our login form.

We need to wire up an endpoint placeholder, just like we did for `GET /`. We will serve the login form at `GET /login`.

```rust
//! src/routes/mod.rs

// [...]
// New module!
mod login;
pub use login::*;
```

```rust
//! src/routes/login/mod.rs
mod get;
pub use get::login_form;
```

```rust
//! src/routes/login/get.rs
use actix_web::HttpResponse;

pub async fn login_form() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

```rust
//! src/startup.rs
use crate::routes::{/* */, login_form};
// [...]

fn run(/* */) -> Result<Server, std::io::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/login", web::get().to(login_form))
            // [...]
    })
    // [...]
}
```

10.6.1 HTML Forms

The HTML will be more convoluted this time:

```html
<!-- src/routes/login/login.html -->
<!DOCTYPE html>
<html lang="en">
    <head>
```

```html
        <meta http-equiv="content-type" content="text/html; charset=utf-8">
        <title>Login</title>
    </head>
    <body>
        <form>
            <label>Username
                <input
                    type="text"
                    placeholder="Enter Username"
                    name="username"
                >
            </label>

            <label>Password
                <input
                    type="password"
                    placeholder="Enter Password"
                    name="password"
                >
            </label>

            <button type="submit">Login</button>
        </form>
    </body>
</html>
```

```rust
//! src/routes/login/get.rs
use actix_web::HttpResponse;
use actix_web::http::header::ContentType;

pub async fn login_form() -> HttpResponse {
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(include_str!("login.html"))
}
```

`form` is the HTML element doing the heavy-lifting here. Its job is to collect a set of data fields and send them over for processing to a backend server.

The fields are defined using the `input` element - we have two here: username and password. Inputs are given a `type` attribute - it tells the browser how to display them.
`text` and `password` will both be rendered as a single-line free-text field, with one key difference: the characters entered into a `password` field are obfuscated.

Each `input` is wrapped in a `label` element:

- clicking on the label name toggles the input field;
- it improves accessibility for screen-readers users (it is read out loud when the user is focused on the element).

On each input we have set two other attributes:

- `placeholder`, whose value is shown as a suggestion within the text field before the user starts filling the form;
- `name`, the key that we must use in the backend to identify the field value within the submitted form data.

At the end of the form, there is a `button` - it will trigger the submission of the provided input to the backend.

What happens if you enter a random username and password and try to submit it?
The page refreshes, the input fields are reset - the URL has changed though!
It should now be `localhost:8000/login?username=myusername&password=mysecretpassword`.

This is `form`'s default behaviour[17] - `form` submits the data to the very same page it is being served from (i.e. `/login`) using the `GET` HTTP verb. This is far from ideal - as you have just witnessed, a form submitted via `GET` encodes all input data in clear text as query parameters. Being part of the URL, they end up stored as part of the browser's navigation history. Query parameters are also captured in logs (e.g. `http.route` property in our own backend).
We really do not want passwords or any type of sensitive data there.

We can change this behaviour by setting a value for `action` and `method` on `form`:

```html
<!-- src/routes/login/login.html -->
<!-- [...] -->
<form action="/login" method="post">
<!-- [...] -->
```

We could technically omit `action`, but the default behaviour is not particularly well-documented therefore it is clearer to define it explicitly.
Thanks to `method="post"` the input data will be passed to the backend using the request body, a much safer option.

If you try to submit the form again, you should see a `404` in the API logs for `POST /login`. Let's define the endpoint!

```rust
//! src/routes/login/mod.rs
// [...]
mod post;
pub use post::login;
```

```rust
//! src/routes/login/post.rs
use actix_web::HttpResponse;

pub async fn login() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

```rust
//! src/startup.rs
use crate::routes::login;
// [...]

fn run(/* */) -> Result</* */> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/login", web::post().to(login))
            // [...]
```

[17] It begs the question of **why** `GET` was chosen as default `method`, considering it is strictly less secure. We also do not see any warnings in the browser's console, even though we are obviously transmitting sensitive data in clear text via query parameters (a field with type `password`, form using `GET` as `method`).

```
    })
    // [...]
}
```

10.6.2 Redirect On Success

Try to log in again: the form will disappear, you will be greeted by a blank page. It is not the best kind of feedback - it would be ideal to show a message confirming that the user has logged in successfully. Furthermore, if the user tries to refresh the page, they will be prompted by the browser to confirm that they want to submit the form again.

We can improve the situation by using a **redirect** - if authentication succeeds, we instruct the browser to navigate back to our home page.

A redirect response requires two elements:

- a redirect status code;
- a `Location` header, set to the URL we want to redirect to.

All redirect status codes are in the `3xx` range - we need to choose the most appropriate one depending on the HTTP verb and the semantic meaning we want to communicate (e.g. temporary vs permanent redirection).
You can find a comprehensive guide on MDN Web Docs. `303 See Other` is the most fitting for our usecase (confirmation page after form submission):

```rust
//! src/routes/login/post.rs
use actix_web::http::header::LOCATION;
use actix_web::HttpResponse;

pub async fn login() -> HttpResponse {
    HttpResponse::SeeOther()
        .insert_header((LOCATION, "/"))
        .finish()
}
```

You should now see `Welcome to our newsletter!` after form submission.

10.6.3 Processing Form Data

Truth be told, we are not redirecting on success - we are **always** redirecting.
We need to enhance our `login` function to actually verify the incoming credentials.

As we have seen in chapter 3, form data is submitted to the backend using the `application/x-www-form-urlencoded` content type.
We can parse it out of the incoming request using `actix-web`'s `Form` extractor and a struct that implements `serde::Deserialize`:

```rust
//! src/routes/login/post.rs
// [...]
use actix_web::web;
use secrecy::Secret;

#[derive(serde::Deserialize)]
pub struct FormData {
    username: String,
    password: Secret<String>,
}

pub async fn login(_form: web::Form<FormData>) -> HttpResponse {
    // [...]
}
```

We built the foundation of password-based authentication in the earlier part of this chapter - let's look again at the auth code in the handler for `POST /newsletters`:

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(
    name = "Publish a newsletter issue",
    skip(body, pool, email_client, request),
    fields(username=tracing::field::Empty, user_id=tracing::field::Empty)
)]
pub async fn publish_newsletter(
    body: web::Json<BodyData>,
    pool: web::Data<PgPool>,
    email_client: web::Data<EmailClient>,
    request: HttpRequest,
) -> Result<HttpResponse, PublishError> {
    let credentials = basic_authentication(request.headers())
        .map_err(PublishError::AuthError)?;
    tracing::Span::current()
        .record("username", &tracing::field::display(&credentials.username));
    let user_id = validate_credentials(credentials, &pool).await?;
    tracing::Span::current()
        .record("user_id", &tracing::field::display(&user_id));
    // [...]
```

`basic_authentication` deals with the extraction of credentials from the `Authorization` header when using the 'Basic' authentication scheme - not something we are interested in reusing in `login`.
`validation_credentials`, instead, is what we are looking for: it takes username and password as input, returning either the corresponding `user_id` (if authentication is successful) or an error (if credentials are invalid).
The current definition of `validation_credentials` is polluted by the concerns of `publish_newsletters`:

```rust
//! src/routes/newsletters.rs
// [...]

async fn validate_credentials(
    credentials: Credentials,
    pool: &PgPool,
    // We are returning a `PublishError`,
    // which is a specific error type detailing
    // the relevant failure modes of `POST /newsletters`
    // (not just auth!)
) -> Result<uuid::Uuid, PublishError> {
    let mut user_id = None;
    let mut expected_password_hash = Secret::new(
        "$argon2id$v=19$m=15000,t=2,p=1$\
        gZiV/M1gPc22ElAH/Jh1Hw$\
        CWOrkoo7oJBQ/iyh7uJ0LO2aLEfrHwTWllSAxT0zRno"
            .to_string()
    );

    if let Some((stored_user_id, stored_password_hash)) =
        get_stored_credentials(&credentials.username, pool)
            .await
            .map_err(PublishError::UnexpectedError)?
    {
        user_id = Some(stored_user_id);
        expected_password_hash = stored_password_hash;
    }

    spawn_blocking_with_tracing(move || {
        verify_password_hash(expected_password_hash, credentials.password)
    })
    .await
    .context("Failed to spawn blocking task.")
    .map_err(PublishError::UnexpectedError)??;

    user_id.ok_or_else(|| PublishError::AuthError(anyhow::anyhow!("Unknown username.")))
}
```

10.6.3.1 Building An `authentication` Module

Let's refactor `validate_credentials` to prepare it for extraction - we want to build a shared `authentication` module, which we will use in both `POST /login` and `POST /newsletters`.
Let's define `AuthError`, a new error enum:

```
//! src/lib.rs
pub mod authentication;
// [...]
```

```
//! src/authentication.rs

#[derive(thiserror::Error, Debug)]
pub enum AuthError {
    #[error("Invalid credentials.")]
    InvalidCredentials(#[source] anyhow::Error),
    #[error(transparent)]
    UnexpectedError(#[from] anyhow::Error),
}
```

We are using an enumeration because, just like we did in `POST /newsletters`, we want to empower the caller to react differently depending on the error type - i.e. return a `500` for `UnexpectedError`, while `AuthError`s should result into a `401`.
Let's change the signature of `validate_credentials` to return `Result<uuid::Uuid, AuthError>` now:

```
//! src/routes/newsletters.rs
use crate::authentication::AuthError;
// [...]

```

```rust
async fn validate_credentials(
    // [...]
) -> Result<uuid::Uuid, AuthError> {
    // [...]

    if let Some(/* */) = get_stored_credentials(/* */).await?
    {/* */}

    spawn_blocking_with_tracing(/* */)
        .await
        .context("Failed to spawn blocking task.")??;

    user_id
        .ok_or_else(|| anyhow::anyhow!("Unknown username."))
        .map_err(AuthError::InvalidCredentials)
}
```

`cargo check` returns two errors now:

```
error[E0277]: `?` couldn't convert the error to `AuthError`
   --> src/routes/newsletters.rs
    |
    |     .context("Failed to spawn blocking task.")??;
    |                                                ^
      the trait `From<PublishError>` is not implemented for `AuthError`

error[E0277]: `?` couldn't convert the error to `PublishError`
   --> src/routes/newsletters.rs
    |
    |     let user_id = validate_credentials(credentials, &pool).await?;
    |                                                                 ^
      the trait `From<AuthError>` is not implemented for `PublishError`
    |
```

The first error comes from `validate_credentials` itself - we are calling `verify_password_hash`, which is still returning a `PublishError`.

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(/* */)]
fn verify_password_hash(
    expected_password_hash: Secret<String>,
    password_candidate: Secret<String>,
) -> Result<(), PublishError> {
    let expected_password_hash = PasswordHash::new(
            expected_password_hash.expose_secret()
        )
        .context("Failed to parse hash in PHC string format.")
        .map_err(PublishError::UnexpectedError)?;

    Argon2::default()
        .verify_password(
            password_candidate.expose_secret().as_bytes(),
            &expected_password_hash
        )
        .context("Invalid password.")
        .map_err(PublishError::AuthError)
}
```

Let's fix it:

```rust
//! src/routes/newsletters.rs
// [...]

#[tracing::instrument(/* */)]
fn verify_password_hash(/* */) -> Result<(), AuthError> {
    let expected_password_hash = PasswordHash::new(/* */)
        .context("Failed to parse hash in PHC string format.")?;

    Argon2::default()
        .verify_password(/* */)
        .context("Invalid password.")
        .map_err(AuthError::InvalidCredentials)
}
```

Let's deal with second error now:

```
error[E0277]: `?` couldn't convert the error to `PublishError`
   --> src/routes/newsletters.rs
    |
    |     let user_id = validate_credentials(credentials, &pool).await?;
    |                                                                  ^
      the trait `From<AuthError>` is not implemented for `PublishError`
    |
```

This comes from the call to `verify_credentials` inside `publish_newsletters`, the request handler.
`AuthError` does not implement a conversion into `PublishError`, therefore the `?` operator cannot be used.
We will call `map_err` to perform the mapping inline:

```rust
//! src/routes/newsletters.rs
// [...]

pub async fn publish_newsletter(/* */) -> Result<HttpResponse, PublishError> {
    // [...]
    let user_id = validate_credentials(credentials, &pool)
        .await
        // We match on `AuthError`'s variants, but we pass the **whole** error
        // into the constructors for `PublishError` variants. This ensures that
        // the context of the top-level wrapper is preserved when the error is
        // logged by our middleware.
        .map_err(|e| match e {
            AuthError::InvalidCredentials(_) => PublishError::AuthError(e.into()),
            AuthError::UnexpectedError(_) => PublishError::UnexpectedError(e.into()),
        })?;
    // [...]
}
```

The code should compile now.
Let's complete the extraction by moving `validate_credentials`, `Credentials`, `get_stored_credentials` and `verify_password_hash` into the `authentication` module:

```rust
//! src/authentication.rs
use crate::telemetry::spawn_blocking_with_tracing;
use anyhow::Context;
use secrecy::{Secret, ExposeSecret};
use argon2::{Argon2, PasswordHash, PasswordVerifier};
use sqlx::PgPool;
// [...]

pub struct Credentials {
    // These two fields were not marked as `pub` before!
    pub username: String,
    pub password: Secret<String>,
}

#[tracing::instrument(/* */)]
pub async fn validate_credentials(/* */) -> Result</* */> {
    // [...]
}

#[tracing::instrument(/* */)]
fn verify_password_hash(/* */) -> Result</* */> {
    // [...]
}

#[tracing::instrument(/* */)]
async fn get_stored_credentials(/* */) -> Result</* */> {
    // [...]
}
```

```rust
//! src/routes/newsletters.rs
// [...]
use crate::authentication::{validate_credentials, AuthError, Credentials};
// There will be warnings about unused imports, follow the compiler to fix them!
// [...]
```

10.6.3.2 Rejecting Invalid Credentials

The extracted `authentication` module is now ready to be used in our `login` function. Let's plug it in:

```rust
//! src/routes/login/post.rs
use crate::authentication::{validate_credentials, Credentials};
use actix_web::http::header::LOCATION;
use actix_web::web;
use actix_web::HttpResponse;
use secrecy::Secret;
use sqlx::PgPool;

#[derive(serde::Deserialize)]
pub struct FormData {
    username: String,
    password: Secret<String>,
}

#[tracing::instrument(
    skip(form, pool),
    fields(username=tracing::field::Empty, user_id=tracing::field::Empty)
)]
// We are now injecting `PgPool` to retrieve stored credentials from the database
pub async fn login(form: web::Form<FormData>, pool: web::Data<PgPool>) -> HttpResponse {
    let credentials = Credentials {
        username: form.0.username,
        password: form.0.password,
    };
    tracing::Span::current()
        .record("username", &tracing::field::display(&credentials.username));
    match validate_credentials(credentials, &pool).await {
        Ok(user_id) => {
            tracing::Span::current()
                .record("user_id", &tracing::field::display(&user_id));
            HttpResponse::SeeOther()
                .insert_header((LOCATION, "/"))
                .finish()
        }
        Err(_) => {
            todo!()
        }
    }
}
```

A login attempt using random credentials should now fail: the request handler panics due to `validation_credentials` returning an error, which in turn leads to `actix-web` dropping the connection. It is not a graceful failure - the browser is likely to show something along the lines of `The connection was reset`.
We should try as much as possible to avoid panics in request handlers - all errors should be handled gracefully.
Let's introduce a `LoginError`:

```rust
//! src/routes/login/post.rs
// [...]
use crate::authentication::AuthError;
use crate::routes::error_chain_fmt;
use actix_web::http::StatusCode;
use actix_web::{web, ResponseError};

#[tracing::instrument(/* */)]
pub async fn login(/* */) -> Result<HttpResponse, LoginError> {
    // [...]
    let user_id = validate_credentials(credentials, &pool)
        .await
        .map_err(|e| match e {
            AuthError::InvalidCredentials(_) => LoginError::AuthError(e.into()),
            AuthError::UnexpectedError(_) => LoginError::UnexpectedError(e.into()),
        })?;
    tracing::Span::current().record("user_id", &tracing::field::display(&user_id));
    Ok(HttpResponse::SeeOther()
        .insert_header((LOCATION, "/"))
        .finish())
}

#[derive(thiserror::Error)]
```

```rust
pub enum LoginError {
    #[error("Authentication failed")]
    AuthError(#[source] anyhow::Error),
    #[error("Something went wrong")]
    UnexpectedError(#[from] anyhow::Error),
}

impl std::fmt::Debug for LoginError {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        error_chain_fmt(self, f)
    }
}

impl ResponseError for LoginError {
    fn status_code(&self) -> StatusCode {
        match self {
            LoginError::UnexpectedError(_) => StatusCode::INTERNAL_SERVER_ERROR,
            LoginError::AuthError(_) => StatusCode::UNAUTHORIZED,
        }
    }
}
```

The code is very similar to what we wrote a few sections ago while refactoring `POST /newsletters`.
What is the effect on the browser?

Submission of the form triggers a page load, resulting in `Authentication failed` being shown on screen[18].
Much better than before, we are making progress!

10.6.4 Contextual Errors

The error message is clear enough - but what should the user do next?
It is reasonable to assume that they want to try to enter their credentials again - they have probably misspelled their username or their password.

We need the error message to appear on top of the login form - providing the user with information while allowing them to quickly retry.

10.6.4.1 Naive Approach

What is the simplest possible approach?
We could return the login HTML page from `ResponseError`, injecting an additional paragraph (`<p>` HTML element) reporting the error to the user.
It would look like this:

```rust
//! src/routes/login/post.rs
// [...]

impl ResponseError for LoginError {
    fn error_response(&self) -> HttpResponse {
        HttpResponse::build(self.status_code())
            .content_type(ContentType::html())
```

[18] The default implementation of `error_response` provided by `actix_web`'s `ResponseError` trait populates the body using the `Display` representation of the error returned by the request handler.

```
            .body(format!(
                r#"<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Login</title>
</head>
<body>
    <p><i>{}</i></p>
    <form action="/login" method="post">
        <label>Username
            <input
                type="text"
                placeholder="Enter Username"
                name="username"
            >
        </label>
        <label>Password
            <input
                type="password"
                placeholder="Enter Password"
                name="password"
            >
        </label>
        <button type="submit">Login</button>
    </form>
</body>
</html>"#,
                self
            ))
    }

    fn status_code(&self) -> StatusCode {
        match self {
            LoginError::UnexpectedError(_) => StatusCode::INTERNAL_SERVER_ERROR,
            LoginError::AuthError(_) => StatusCode::UNAUTHORIZED,
        }
    }
}
```

This approach has a few drawbacks:

- we have two slightly-different-but-almost-identical login pages, defined in two different places. If we decide to make changes to the login form, we need to remember to edit both;
- the user is prompted to confirm form resubmission if they try to refresh the page after an unsuccessful login attempt.

To solve the second issue, we need the user to land on a `GET` endpoint.
To solve the first issue, we need to find a way to reuse the HTML we wrote in `GET /login`, instead of duplicating it.

We can achieve both goals with another redirect: if authentication fails, we send the user back to `GET /login`.

```
//! src/routes/login/post.rs
// [...]
```

```rust
impl ResponseError for LoginError {
    fn error_response(&self) -> HttpResponse {
        HttpResponse::build(self.status_code())
            .insert_header((LOCATION, "/login"))
            .finish()
    }

    fn status_code(&self) -> StatusCode {
        StatusCode::SEE_OTHER
    }
}
```

Unfortunately a vanilla redirect is not enough - the browser would show the login form to the user again, with no feedback explaining that their login attempt was unsuccessful.
We need to find a way to instruct `GET /login` to show an error message.

Let's explore a few options.

10.6.4.2 Query Parameters

The value of the `Location` header determines the URL the user will be redirected to.
It does not end there though - we can also specify query parameters!

Let's encode the authentication error message in an `error` query parameter.
Query parameters are part of the URL - therefore we need to URL-encode the display representation of `LoginError`.

```toml
#! Cargo.toml
# [...]
[dependencies]
urlencoding = "2"
# [...]
```

```rust
//! src/routes/login/post.rs
// [...]

impl ResponseError for LoginError {
    fn error_response(&self) -> HttpResponse {
        let encoded_error = urlencoding::Encoded::new(self.to_string());
        HttpResponse::build(self.status_code())
            .insert_header((LOCATION, format!("/login?error={}", encoded_error)))
            .finish()
    }
    // [...]
}
```

The `error` query parameter can then be extracted in the request handler for `GET /login`.

```rust
//! src/routes/login/get.rs
use actix_web::{web, HttpResponse, http::header::ContentType};

#[derive(serde::Deserialize)]
pub struct QueryParams {
    error: Option<String>,
}
```

```rust
pub async fn login_form(query: web::Query<QueryParams>) -> HttpResponse {
    let _error = query.0.error;
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(include_str!("login.html"))
}
```

Finally, we can customise the returned HTML page based on its value:

```rust
//! src/routes/login/get.rs
// [...]

pub async fn login_form(query: web::Query<QueryParams>) -> HttpResponse {
    let error_html = match query.0.error {
        None => "".into(),
        Some(error_message) => format!("<p><i>{error_message}</i></p>"),
    };
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(
            r#"<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Login</title>
</head>
<body>
    {error_html}
    <form action="/login" method="post">
        <label>Username
            <input
                type="text"
                placeholder="Enter Username"
                name="username"
            >
        </label>
        <label>Password
            <input
                type="password"
                placeholder="Enter Password"
                name="password"
            >
        </label>
        <button type="submit">Login</button>
    </form>
</body>
</html>"#,
        ))
}
```

It works[19]!

[19] Our web pages are not particularly dynamic - we are mostly looking at the injection of a few elements, `format!` does the job without breaking a sweat. The same approach does not scale very well when working on more complex user interfaces - you will need to build reusable components to be shared across multiple pages while performing loops and conditionals on many different pieces of dynamic data. Template engines are a common approach to handle this new level of complexity - `tera` and `askama` are popular options in the Rust ecosystem.

10.6.4.3 Cross-Site Scripting (XSS)

Query parameters are not private - our backend server cannot prevent users from tweaking the URL.
In particular, it cannot prevent attackers from playing with them.

Try to navigate to the following URL:

```
http://localhost:8000/login?error=Your%20account%20has%20been%20locked%2C%20
  please%20submit%20your%20details%20%3Ca%20href%3D%22https%3A%2F%2Fzero2prod.com
   %22%3Ehere%3C%2Fa%3E%20to%20resolve%20the%20issue.
```

On top of the login form you will see

> Your account has been locked, please submit your details *here* to resolve the issue.

with `here` being a link to another website (zero2prod.com, in this case).
In a more realistic scenario, `here` would link to a website controlled by an attacker, luring the victim into disclosing their login credentials.

This is known as a **cross-site scripting** attack (XSS).
The attacker injects HTML fragments or JavaScript snippets into a trusted website by exploiting dynamic content built from untrusted sources - e.g. user inputs, query parameters, etc.
From a user perspective, XSS attacks are particularly insidious - the URL matches the one you wanted to visit, therefore you are likely to trust the displayed content.

OWASP provides an extensive cheat sheet on how to prevent XSS attacks - I strongly recommend familiarising with it if you are working on a web application.

Let's look at the guidance for our issue here: we want to display untrusted data (the value of a query parameter) inside an HTML element (`<p><i>UNTRUSTED DATA HERE</i></p>`).
According to OWASP's guidelines, we must HTML entity-encode the untrusted input - i.e.:

- convert `&` to `&amp;`
- convert `<` to `&lt;`
- convert `>` to `&gt;`
- convert `"` to `&quot;`
- convert `'` to `&#x27;`
- convert `/` to `&#x2F`.

HTML entity encoding prevents the insertion of further HTML elements by escaping the characters required to define them.
Let's amend our `login_form` handler:

```
#! Cargo.toml
# [...]
[dependencies]
htmlescape = "0.3"
# [...]
```

```
//! src/routes/login/get.rs
// [...]
```

```rust
pub async fn login_form(/* */) -> HttpResponse {
    let error_html = match query.0.error {
        None => "".into(),
        Some(error_message) => format!(
            "<p><i>{}</i></p>",
            htmlescape::encode_minimal(&error_message)
        ),
    };
    // [...]
}
```

Load the compromised URL again - you will see a different message:

> Your account has been locked, please submit your details <a href=“https://zero2prod.com”>here</a> to resolve the issue.

The HTML a element is no longer rendered by the browser - the user has now reasons to suspect that something fishy is going on.
Is it enough?

At the very least, users are less likely to copy-paste and navigate to the link compared to just clicking on `here`. Nonetheless, attackers are not naive - they will amend the injected message as soon as they notice that our website is performing HTML entity encoding. It could be as simple as

> Your account has been locked, please call +CC3332288777 to resolve the issue.

This might be good enough to lure in a couple of victims. We need something stronger than character escaping.

10.6.4.4 Message Authentication Codes

We need a mechanism to verify that the query parameters have been set by our API and that they have not been altered by a third party.
This is known as **message authentication** - it guarantees that the message has not been modified in transit (**integrity**) and it allows you to verify the identity of the sender (**data origin authentication**).

Message authentication codes (MACs) are a common technique to provide message authentication - a *tag* is added to the message allowing verifiers to check its integrity and origin.
HMAC are a well-known family of MACs - **h**ash-based **m**essage **a**uthentication **c**odes.

HMACs are built around a secret and a hash function.
The secret is prepended to the message and the resulting string is fed into the hash function. The resulting hash is then concatenated to the secret and hashed again - the output is the message tag.
In pseudo-code:

```rust
let hmac_tag = hash(
    concat(
        key,
        hash(concat(key, message))
    )
);
```

We are deliberately omitting a few nuances around key padding - you can find all the details in RFC 2104.

10.6.4.5 Add An HMAC Tag To Protect Query Parameters

Let's try to use a HMAC to verify integrity and provenance for our query parameters.
The Rust Crypto organization provides an implementation of HMACs, the `hmac` crate. We will also need a hash function - let's go for SHA-256.

```toml
#! Cargo.toml
# [...]
[dependencies]
hmac = { version = "0.12", features = ["std"] }
sha2 = "0.10"
# [...]
```

Let's add another query parameter to our `Location` header, `tag`, to store the HMAC of our error message.

```rust
//! src/routes/login/post.rs
use hmac::{Hmac, Mac};
// [...]

impl ResponseError for LoginError {
    fn error_response(&self) -> HttpResponse {
        let query_string = format!(
            "error={}",
            urlencoding::Encoded::new(self.to_string())
        );
        // We need the secret here - how do we get it?
        let secret: &[u8] = todo!();
        let hmac_tag = {
            let mut mac = Hmac::<sha2::Sha256>::new_from_slice(secret).unwrap();
            mac.update(query_string.as_bytes());
            mac.finalize().into_bytes()
        };
        HttpResponse::build(self.status_code())
            // Appending the hexadecimal representation of the HMAC tag to the
            // query string as an additional query parameter.
            .insert_header((
                LOCATION,
                format!("/login?{query_string}&tag={hmac_tag:x}")
            ))
            .finish()
    }
    // [...]
}
```

The code snippet is *almost* perfect - we just need a way to get our secret!
Unfortunately it will not be possible from within `ResponseError` - we only have access to the error type (`LoginError`) that we are trying to convert into an HTTP response. `ResponseError` is just a specialised `Into` trait.
In particular, we do not have access to the application state (i.e. we cannot use the `web::Data` extractor), which is where we would be storing the secret.

Let’s move our code back into the request handler:

```rust
//! src/routes/login/post.rs
use secret::ExposeSecret;
// [...]

#[tracing::instrument(
    skip(form, pool, secret),
    fields(username=tracing::field::Empty, user_id=tracing::field::Empty)
)]
pub async fn login(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
    // Injecting the secret as a secret string for the time being.
    secret: web::Data<Secret<String>>,
    // No longer returning a `Result<HttpResponse, LoginError>`!
) -> HttpResponse {
    // [...]
    match validate_credentials(credentials, &pool).await {
        Ok(user_id) => {
            tracing::Span::current()
                .record("user_id", &tracing::field::display(&user_id));
            HttpResponse::SeeOther()
                .insert_header((LOCATION, "/"))
                .finish()
        }
        Err(e) => {
            let e = match e {
                AuthError::InvalidCredentials(_) => LoginError::AuthError(e.into()),
                AuthError::UnexpectedError(_) => LoginError::UnexpectedError(e.into()),
            };
            let query_string = format!(
                "error={}",
                urlencoding::Encoded::new(e.to_string())
            );
            let hmac_tag = {
                let mut mac = Hmac::<sha2::Sha256>::new_from_slice(
                    secret.expose_secret().as_bytes()
                ).unwrap();
                mac.update(query_string.as_bytes());
                mac.finalize().into_bytes()
            };
            HttpResponse::SeeOther()
                .insert_header((
                    LOCATION,
                    format!("/login?{}&tag={:x}", query_string, hmac_tag),
                ))
                .finish()
        }
    }
}

// The `ResponseError` implementation for `LoginError` has been deleted.
```

This is a viable approach - and it compiles.
It has one drawback - we are no longer propagating upstream, to the middleware chain, the error context. This is concerning when dealing with a `LoginError::UnexpectedError` - our logs

should really capture what has gone wrong.
Luckily enough, there is a way to have our cake and eat it too: `actix_web::error::InternalError`.
`InternalError` can be built from a `HttpResponse` and an error. It can be returned as an error from a request handler (it implements `ResponseError`) and it returns to the caller the `HttpResponse` you passed to its constructor - exactly what we needed!

Let's change `login` once again to use it:

```rust
//! src/routes/login/post.rs
// [...]
use actix_web::error::InternalError;

#[tracing::instrument(/* */)]
// Returning a `Result` again!
pub async fn login(/* */) -> Result<HttpResponse, InternalError<LoginError>> {
    // [...]
    match validate_credentials(credentials, &pool).await {
        Ok(user_id) => {
            // [...]
            // We need to Ok-wrap again
            Ok(/* */)
        }
        Err(e) => {
            // [...]
            let response = HttpResponse::SeeOther()
                .insert_header((
                    LOCATION,
                    format!("/login?{}&tag={:x}", query_string, hmac_tag),
                ))
                .finish();
            Err(InternalError::from_response(e, response))
        }
    }
}
```

Error reporting has been saved.
One last task left for us: injecting the secret used by our HMACs into the application state.

```rust
//! src/configuration.rs
// [...]
#[derive(serde::Deserialize, Clone)]
pub struct ApplicationSettings {
    // [...]
    pub hmac_secret: Secret<String>
}
```

```rust
//! src/startup.rs
use secrecy::Secret;
// [...]

impl Application {
    pub async fn build(configuration: Settings) -> Result<Self, std::io::Error> {
        // [...]
        let server = run(
            // [...]
            configuration.application.hmac_secret,
        )?;
        // [...]
    }
}

fn run(
    // [...]
    hmac_secret: Secret<String>,
) -> Result<Server, std::io::Error> {
    let server = HttpServer::new(move || {
        // [...]
        .app_data(Data::new(hmac_secret.clone()))
    })
    // [...]
}
```

```yaml
#! configuration/base.yml
application:
    # [...]
    # You need to set the `APP_APPLICATION__HMAC_SECRET` environment variable
    # on Digital Ocean as well for production!
    hmac_secret: "super-long-and-secret-random-key-needed-to-verify-message-integrity"
# [...]
```

Using `Secret<String>` as the type injected into the application state is far from ideal. `String` is a primitive type and there is a significant risk of conflict - i.e. another middleware or service registering another `Secret<String>` against the application state, overriding our HMAC secret (or vice versa).
Let's create a wrapper type to sidestep the issue:

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

fn run(
    // [...]
    hmac_secret: HmacSecret,
) -> Result<Server, std::io::Error> {
    let server = HttpServer::new(move || {
        // [...]
        .app_data(Data::new(HmacSecret(hmac_secret.clone())))
    })
    // [...]
}

#[derive(Clone)]
pub struct HmacSecret(pub Secret<String>);
```

```rust
//! src/routes/login/post.rs
use crate::startup::HmacSecret;
// [...]

#[tracing::instrument(/* */)]
pub async fn login(
    // [...]
    // Inject the wrapper type!
    secret: web::Data<HmacSecret>,
) -> Result<HttpResponse, InternalError<LoginError>> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(/* */) => {/* */}
        Err(/* */) => {
            // [...]
            let hmac_tag = {
                let mut mac = Hmac::<sha2::Sha256>::new_from_slice(
                    secret.0.expose_secret().as_bytes()
                ).unwrap();
                // [...]
            };
            // [...]
        }
    }
}
```

10.6.4.6 Verifying The HMAC Tag

Time to validate that tag in `GET /login`!

Let's start by extracting the `tag` query parameter.
We are currently using the `Query` extractor to parse the incoming query parameters into a `QueryParams` struct, which features an optional `error` field.
Going forward, we foresee two scenarios:

- There is no error (e.g. you just landed on the login page), therefore we do not expect any query parameter;
- There is an error to be reported, therefore we expect to see both an `error` and a `tag` query parameter.

Changing `QueryParams` from

```rust
#[derive(serde::Deserialize)]
pub struct QueryParams {
    error: Option<String>,
}
```

to

```rust
#[derive(serde::Deserialize)]
pub struct QueryParams {
    error: Option<String>,
    tag: Option<String>,
}
```

would not capture the new requirements accurately - it would allow callers to pass a `tag` parameter while omitting the `error` one, or vice versa. We would need to do extra validation in the request handler to make sure this is not the case.
We can avoid this issue entirely by making all fields in `QueryParams` required while `QueryParams` itself becomes optional:

```rust
//! src/routes/login/get.rs
// [...]

```

```rust
#[derive(serde::Deserialize)]
pub struct QueryParams {
    error: String,
    tag: String,
}

pub async fn login_form(query: Option<web::Query<QueryParams>>) -> HttpResponse {
    let error_html = match query {
        None => "".into(),
        Some(query) => {
            format!("<p><i>{}</i></p>", htmlescape::encode_minimal(&query.0.error))
        }
    };
    // [...]
}
```

A neat little reminder to make illegal state impossible to represent using types!
To verify the tag we will need access to the HMAC shared secret - let's inject it:

```rust
//! src/routes/login/get.rs
use crate::startup::HmacSecret;
// [...]

pub async fn login_form(
    query: Option<web::Query<QueryParams>>,
    secret: web::Data<HmacSecret>,
) -> HttpResponse {
    // [...]
}
```

`tag` was a byte slice encoded as a hex string. We will need the `hex` crate to decode it back to bytes in `GET /login`. Let's add it as a dependency:

```toml
#! Cargo.toml
# [...]
[dependencies]
# [...]
hex = "0.4"
```

We can now define a `verify` method on `QueryParams` itself: it will return the error string if the message authentication code matches our expectations, an error otherwise.

```rust
//! src/routes/login/get.rs
use hmac::{Hmac, Mac};
use secrecy::ExposeSecret;
// [...]

impl QueryParams {
    fn verify(self, secret: &HmacSecret) -> Result<String, anyhow::Error> {
        let tag = hex::decode(self.tag)?;
        let query_string = format!("error={}", urlencoding::Encoded::new(&self.error));

        let mut mac = Hmac::<sha2::Sha256>::new_from_slice(
            secret.0.expose_secret().as_bytes()
        ).unwrap();
        mac.update(query_string.as_bytes());
        mac.verify_slice(&tag)?;

        Ok(self.error)
    }
}
```

We now need to amend the request handler to call it, which raises a question: what do we want to do if the verification fails?
One approach is to fail the entire request by returning a `400`. Alternatively, we can log the verification failure as a warning and skip the error message when rendering the HTML.

Let's go for the latter - a user being redirected with some dodgy query parameters will see our login page, an acceptable scenario.

```
//! src/routes/login/get.rs
// [...]

pub async fn login_form(/* */) -> HttpResponse {
    let error_html = match query {
        None => "".into(),
        Some(query) => match query.0.verify(&secret) {
            Ok(error) => {
                format!("<p><i>{}</i></p>", htmlescape::encode_minimal(&error))
            }
            Err(e) => {
                tracing::warn!(
                    error.message = %e,
                    error.cause_chain = ?e,
                    "Failed to verify query parameters using the HMAC tag"
                );
                "".into()
            }
        },
    };
    // [...]
}
```

You can try again to load our scammy URL:

```
http://localhost:8000/login?error=Your%20account%20has%20been%20locked%2C%20
   please%20submit%20your%20details%20%3Ca%20href%3D%22https%3A%2F%2Fzero2prod.com
     %22%3Ehere%3C%2Fa%3E%20to%20resolve%20the%20issue.
```

No error message should be rendered by the browser!

10.6.4.7 Error Messages Must Be Ephemeral

Implementation-wise, we are happy: the error is rendered as expected and nobody can tamper with our messages thanks to the HMAC tag. Should we deploy it?

We chose a query parameter to pass along the error message because query parameters are a part of the URL - it is easy to pass them along in the value of the `Location` header when redirecting back to the login form on failures. This is both their strength and their weakness: URLs are stored in the browser history, which is in turn used to provide autocomplete suggestions when typing a URL into the address bar. You can experiment with this yourself: try to type `localhost:8000` in

your address bar - what are the suggestions?
Most of them will be URLs including the `error` query parameter due to all the experiments we have been doing so far. If you pick one with a valid `tag`, the login form will feature an `Authentication failed` error message... even though it has been a while since your last login attempt. This is undesirable.

We would like the error message to be **ephemeral**.
It is shown right after a failed login attempt, but it is not stored in your browser history. The only way to trigger the error message again should be to... fail to log in one more time.

We established that query parameters do not meet our requirements. Do we have other options?
Yes, **cookies**!

> This is a great moment to take a break, this is a long chapter!
> Check out the project snapshot on GitHub if you want to check your implementation.

10.6.4.8 What Is A Cookie?

MDN Web Docs defines an HTTP cookie as

> [...] a small piece of data that a server sends to a user's web browser. The browser may store the cookie and send it back to the same server with later requests.

We can use cookies to implement the same strategy we tried with query parameters:

- The user enters invalid credentials and submits the form;
- `POST /login` sets a cookie containing the error message and redirects the user back to `GET /login`;
- The browser calls `GET /login`, including the values of the cookies currently set for the user;
- `GET /login`'s request handler checks the cookies to see if there is an error message to be rendered;
- `GET /login` returns the HTML form to the caller and deletes the error message from the cookie.

The URL is never touched - all error-related information is exchanged via a side-channel (cookies), invisible to the browser history. The last step in the algorithm ensures that the error message is indeed ephemeral - the cookie is "consumed" when the error message is rendered. If the page is reloaded, the error message will not be shown again.

One-time notifications, the technique we just described, are known as **flash messages**.

10.6.4.9 An Integration Test For Login Failures

So far we have experimented quite freely - we wrote some code, launched the application, played around with it.
We are now approaching the final iteration of our design and it would be nice to capture the desired behaviour using some black-box tests, as we did so far for all the user flows supported by our project.
Writing a test will also help us to get familiar with cookies and their behaviour.

We want to verify what happens on login failures, the topic we have been obsessing over for a few sections now.
Let's start by adding a new `login` module to our test suite:

```rust
//! tests/main.rs
// [...]
mod login;
```

```rust
//! tests/api/login.rs
// Empty for now
```

We will need to send a `POST /login` request - let's add a little helper to our `TestApp`, the HTTP client used to interact with our application in our tests:

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    pub async fn post_login<Body>(&self, body: &Body) -> reqwest::Response
    where
        Body: serde::Serialize,
    {
        reqwest::Client::new()
            .post(&format!("{}/login", &self.address))
            // This `reqwest` method makes sure that the body is URL-encoded
            // and the `Content-Type` header is set accordingly.
            .form(body)
            .send()
            .await
            .expect("Failed to execute request.")
    }

    // [...]
}
```

We can now start to sketch our test case. Before touching cookies, we will begin with a simple assertion - it returns a redirect, status code `303`.

```rust
//! tests/api/login.rs
use crate::helpers::spawn_app;

#[tokio::test]
async fn an_error_flash_message_is_set_on_failure() {
    // Arrange
    let app = spawn_app().await;

    // Act
    let login_body = serde_json::json!({
        "username": "random-username",
        "password": "random-password"
    });
    let response = app.post_login(&login_body).await;

    // Assert
    assert_eq!(response.status().as_u16(), 303);
}
```

The test fails!

```
---- login::an_error_flash_message_is_set_on_failure stdout ----
thread 'login::an_error_flash_message_is_set_on_failure' panicked at
'assertion failed: `(left == right)`
  left: `200`,
 right: `303`'
```

Our endpoint *already* returns a `303` - both in case of failure and success! What is going on?
The answer can be found in `reqwest`'s documentation:

> By default, a `Client` will automatically handle HTTP redirects, having a maximum redirect chain of 10 hops. To customize this behavior, a `redirect::Policy` can be used with a `ClientBuilder`.

`reqwest::Client` sees the `303` status code and automatically proceeds to call `GET /login`, the path specified in the `Location` header, which return a `200` - the status code we see in the assertion panic message.
For the purpose of our testing, we do not want `reqwest::Client` to follow redirects - let's customise the HTTP client behaviour by following the guidance provided in its documentation:

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    pub async fn post_login<Body>(&self, body: &Body) -> reqwest::Response
        where
            Body: serde::Serialize,
    {
        reqwest::Client::builder()
            .redirect(reqwest::redirect::Policy::none())
            .build()
            .unwrap()
            // [...]
    }
    // [...]
}
```

The test should pass now.
We can go one step further - inspect the value of the `Location` header.

```rust
//! tests/api/helpers.rs
// [...]

// Little helper function - we will be doing this check several times throughout
// this chapter and the next one.
pub fn assert_is_redirect_to(response: &reqwest::Response, location: &str) {
    assert_eq!(response.status().as_u16(), 303);
    assert_eq!(response.headers().get("Location").unwrap(), location);
}
```

```rust
//! tests/api/login.rs
use crate::helpers::assert_is_redirect_to;
// [...]

#[tokio::test]
```

```
async fn an_error_flash_message_is_set_on_failure() {
    // [...]

    // Assert
    assert_is_redirect_to(&response, "/login");
}
```

You should see another failure:

```
---- login::an_error_flash_message_is_set_on_failure stdout ----
thread 'login::an_error_flash_message_is_set_on_failure' panicked at
'assertion failed: `(left == right)`
  left: `"/login?error=Authentication%20failed.&tag=2f8fff5[...]"`,
 right: `"/login"`'
```

The endpoint is still using query parameters to pass along the error message. Let's remove that functionality from the request handler:

```
//! src/routes/login/post.rs
// A few imports are now unused and can be removed.
// [...]

#[tracing::instrument(/* */)]
pub async fn login(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
    // We no longer need `HmacSecret`!
) -> Result<HttpResponse, InternalError<LoginError>> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(/* */) => {/* */}
        Err(e) => {
            let e = match e {
                AuthError::InvalidCredentials(_) => LoginError::AuthError(e.into()),
                AuthError::UnexpectedError(_) => LoginError::UnexpectedError(e.into()),
            };
            let response = HttpResponse::SeeOther()
                .insert_header((LOCATION, "/login"))
                .finish();
            Err(InternalError::from_response(e, response))
        }
    }
}
```

I know, it feels like we are going backwards - you need have a bit of patience!
The test should pass. We can now start looking at cookies, which begs the question - what does "set a cookie" *actually* mean?

Cookies are set by attaching a special HTTP header to the response - `Set-Cookie`.
In its simplest form it looks like this:

```
Set-Cookie: <cookie-name>=<cookie-value>
```

`Set-Cookie` can be specified multiple times - one for each cookie you want to set.
`reqwest` provides the `get_all` method to deal with multi-value headers:

```rust
//! tests/api/login.rs
// [...]
use reqwest::header::HeaderValue;
use std::collections::HashSet;

#[tokio::test]
async fn an_error_flash_message_is_set_on_failure() {
    // [...]
    let cookies: HashSet<_> = response
        .headers()
        .get_all("Set-Cookie")
        .into_iter()
        .collect();
    assert!(cookies
        .contains(&HeaderValue::from_str("_flash=Authentication failed").unwrap())
    );
}
```

Truth be told, cookies are so ubiquitous to deserve a dedicated API, sparing us the pain of working with the raw headers. `reqwest` locks this functionality behind the `cookies` feature-flag - let's enable it:

```toml
#! Cargo.toml
# [...]
# Using multi-line format for brevity
[dependencies.reqwest]
version = "0.11"
default-features = false
features = ["json", "rustls-tls", "cookies"]
```

```rust
//! tests/api/login.rs
// [...]
use reqwest::header::HeaderValue;
use std::collections::HashSet;

#[tokio::test]
async fn an_error_flash_message_is_set_on_failure() {
    // [...]
    let flash_cookie = response.cookies().find(|c| c.name() == "_flash").unwrap();
    assert_eq!(flash_cookie.value(), "Authentication failed");
}
```

As you can see, the cookie API is significantly more ergonomic. Nonetheless there is value in touching directly what it abstracts away, at least once.
The test should fail, as expected.

10.6.4.10 How To Set A Cookie In `actix-web`

How do we set a cookie on the outgoing response in `actix-web`?
We can work with headers directly:

```rust
//! src/routes/login/post.rs
// [...]

#[tracing::instrument(/* */)]
```

```rust
pub async fn login(/* */) -> Result<HttpResponse, InternalError<LoginError>> {
    match validate_credentials(/* */).await {
        Ok(/* */) => {/* */}
        Err(e) => {
            // [...]
            let response = HttpResponse::SeeOther()
                .insert_header((LOCATION, "/login"))
                .insert_header(("Set-Cookie", format!("_flash={e}")))
                .finish();
            Err(InternalError::from_response(e, response))
        }
    }
}
```

This change should be enough to get the test to pass.
Just like `reqwest`, `actix-web` provides a dedicated cookie API. `Cookie::new` takes two arguments - the name and the value of the cookie. Let's use it:

```rust
//! src/routes/login/post.rs
use actix_web::cookie::Cookie;
// [...]

#[tracing::instrument(/* */)]
pub async fn login(/* */) -> Result<HttpResponse, InternalError<LoginError>> {
    match validate_credentials(/* */).await {
        Ok(/* */) => {/* */}
        Err(e) => {
            // [...]
            let response = HttpResponse::SeeOther()
                .insert_header((LOCATION, "/login"))
                .cookie(Cookie::new("_flash", e.to_string()))
                .finish();
            Err(InternalError::from_response(e, response))
        }
    }
}
```

The test should stay green.

10.6.4.11 An Integration Test For Login Failures - Part 2

Let's focus on the other side of the story now - `GET /login`. We want to verify that the error message, passed along in the `_flash` cookie, is actually rendered above the login form shown to the user after the redirect.

Let's start by adding a `get_login_html` helper method on `TestApp`:

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // Our tests will only look at the HTML page, therefore
    // we do not expose the underlying reqwest::Response
    pub async fn get_login_html(&self) -> String {
        reqwest::Client::new()
            .get(&format!("{}/login", &self.address))
            .send()
            .await
            .expect("Failed to execute request.")
            .text()
            .await
            .unwrap()
    }
    // [...]
}
```

We can then extend our existing test to call `get_login_html` after having submitted invalid credentials to `POST /login`:

```
//! tests/api/login.rs
// [...]

#[tokio::test]
async fn an_error_flash_message_is_set_on_failure() {
    // [...]
    // Act
    let login_body = serde_json::json!({
        "username": "random-username",
        "password": "random-password"
    });
    let response = app.post_login(&login_body).await;

    // Assert
    // [...]

    // Act - Part 2
    let html_page = app.get_login_html().await;
    assert!(html_page.contains(r#"<p><i>Authentication failed</i></p>"#));
}
```

The test should fail.
As it stands, there is no way for us to get it to pass: we are not propagating the cookies set by `POST /login` when sending a request to `GET /login` - the browser would be expected to fulfill this task in normal circumstances. Can `reqwest` take care of it?

By default, it does not - but it can be configured to! We just need to pass `true` to `reqwest::ClientBuilder::cookie_store`.
There is a caveat though - we must use the same instance of `reqwest::Client` for all requests to our API if we want cookie propagation to work. This requires a bit of refactoring in `TestApp` - we are currently creating a new `reqwest::Client` instance inside every helper method. Let's change `TestApp::spawn_app` to create and store an instance of `reqwest::Client` which we will in turn use in all its helper methods.

```
//! tests/api/helpers.rs
// [...]

pub struct TestApp {
    // [...]
    // New field!
    pub api_client: reqwest::Client
}
```

```rust
pub async fn spawn_app() -> TestApp {
    // [...]
    let client = reqwest::Client::builder()
        .redirect(reqwest::redirect::Policy::none())
        .cookie_store(true)
        .build()
        .unwrap();

    let test_app = TestApp {
        // [...]
        api_client: client,
    };
    // [...]
}


impl TestApp {
    pub async fn post_subscriptions(/* */) -> reqwest::Response {
        self.api_client
            .post(/* */)
            // [...]
    }

    pub async fn post_newsletters(/* */) -> reqwest::Response {
        self.api_client
            .post(/* */)
        // [...]
    }

    pub async fn post_login<Body>(/* */) -> reqwest::Response
        where
            Body: serde::Serialize,
    {
        self.api_client
            .post(/* */)
            // [...]
    }

    pub async fn get_login_html(/* */) -> String {
        self.api_client
            .get(/* */)
        // [...]
    }
    // [...]
}
```

Cookie propagation should now work as expected.

10.6.4.12 How To Read A Cookie In `actix-web`

It's time to look again at our request handler for `GET /login`:

```rust
//! src/routes/login/get.rs
use crate::startup::HmacSecret;
use actix_web::http::header::ContentType;
use actix_web::{web, HttpResponse};
use hmac::{Hmac, Mac, NewMac};

#[derive(serde::Deserialize)]
pub struct QueryParams {
    error: String,
    tag: String,
}

impl QueryParams {
    fn verify(self, secret: &HmacSecret) -> Result<String, anyhow::Error> {
        /* */
    }
}

pub async fn login_form(
    query: Option<web::Query<QueryParams>>,
    secret: web::Data<HmacSecret>,
) -> HttpResponse {
    let error_html = match query {
        None => "".into(),
        Some(query) => match query.0.verify(&secret) {
            Ok(error) => {
                format!("<p><i>{}</i></p>", htmlescape::encode_minimal(&error))
            }
            Err(e) => {
                tracing::warn!(/* */);
                "".into()
            }
        },
    };
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(/* HTML */,))
}
```

Let's begin by ripping out all the code related to query parameters and their (cryptographic) validation:

```rust
//! src/routes/login/get.rs
use actix_web::http::header::ContentType;
use actix_web::HttpResponse;

pub async fn login_form() -> HttpResponse {
    let error_html: String = todo!();
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(/* HTML */))
}
```

Back to the basics. Let's seize this opportunity to remove the dependencies we added during our HMAC adventure - `sha2`, `hmac` and `hex`.

To access cookies on an incoming request we need to get our hands on `HttpRequest` itself. Let's add it as an input to `login_form`:

```rust
//! src/routes/login/get.rs
// [...]
use actix_web::HttpRequest;

pub async fn login_form(request: HttpRequest) -> HttpResponse {
    // [...]
}
```

We can then use `HttpRequest::cookie` to retrieve a cookie given its name:

```rust
//! src/routes/login/get.rs
// [...]

pub async fn login_form(request: HttpRequest) -> HttpResponse {
    let error_html = match request.cookie("_flash") {
        None => "".into(),
        Some(cookie) => {
            format!("<p><i>{}</i></p>", cookie.value())
        }
    };
    // [...]
}
```

Our integration test should pass now!

10.6.4.13 How To Delete A Cookie In `actix-web`

What happens if you refresh the page after a failed login attempt? The error message is still there! The same thing happens if you open a new tab and navigate straight to `localhost:8000/login` - `Authentication failed` will show up on top of the login form.

This is not what we had in mind when we said that error messages should be ephemeral. How do we fix it? There is no `Unset-cookie` header - how do we delete the `_flash` cookie from the user's browser?

Let's zoom in on the lifecycle of a cookie.
When it comes to durability, there are two types of cookies: **session cookies** and **persistent cookies**. Session cookies are stored in memory - they are deleted when the session ends (i.e. the browser is closed). Persistent cookies, instead, are saved to disk and will still be there when you re-open the browser.

A vanilla `Set-Cookie` header creates a session cookie. To set a persistent cookie you must specify an expiration policy using a cookie attribute - either `Max-Age` or `Expires`.
`Max-Age` is interpreted as the number of seconds remaining until the cookie expires - e.g. `Set-Cookie: _flash=omg; Max-Age=5` creates a persistent `_flash` cookie that will be valid for the next 5 seconds.
`Expires`, instead, expects a date - e.g. `Set-Cookie: _flash=omg; Expires=Thu, 31 Dec 2022 23:59:59 GMT;` creates a persistent cookie that will be valid until the end of 2022.

Setting `Max-Age` to 0 instructs the browser to immediately expire the cookie - i.e. to unset it, which is exactly what we want! A bit hacky? Yes, but it is what it is.

Let's kick-off the implementation work. We can start by modifying our integration test to account for this scenario - the error message should **not** be shown if we reload the login page after the first redirect:

```rust
//! tests/api/login.rs
// [...]

#[tokio::test]
async fn an_error_flash_message_is_set_on_failure() {
    // Arrange
    // [...]
    // Act - Part 1 - Try to login
    // [...]
    // Act - Part 2 - Follow the redirect
    // [...]
    // Act - Part 3 - Reload the login page
    let html_page = app.get_login_html().await;
    assert!(!html_page.contains(r#"<p><i>Authentication failed</i></p>"#));
}
```

`cargo test` should report a failure. We now need to change our request handler - we must set the `_flash` cookie on the response with `Max-Age=0` to remove the flash messages stored in the user's browser.:

```rust
//! src/routes/login/get.rs
use actix_web::cookie::{Cookie, time::Duration};
//! [...]

pub async fn login_form(request: HttpRequest) -> HttpResponse {
    // [...]
    HttpResponse::Ok()
        .content_type(ContentType::html())
        .cookie(
            Cookie::build("_flash", "")
                .max_age(Duration::ZERO)
                .finish(),
        )
        .body(/* */)
}
```

The test should pass now!
We can make our intent clearer by refactoring our handler to use the `add_removal_cookie` method:

```rust
//! src/routes/login/get.rs
use actix_web::cookie::{Cookie, time::Duration};
//! [...]

pub async fn login_form(request: HttpRequest) -> HttpResponse {
    // [...]
    let mut response = HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(/* */);
    response
        .add_removal_cookie(&Cookie::new("_flash", ""))
        .unwrap();
    response
}
```

Under the hood, it performs the exact same operation but it does not require the reader to piece together the meaning of setting `Max-Age` to zero.

10.6.4.14 Cookie Security

What security challenges do we face when working with cookies?

It is still possible to perform a XSS attack using cookies, but it requires a bit more effort compared to query parameters - you cannot craft a link to our website that sets or manipulates cookies. Nonetheless, using cookies naively *can* expose us to bad actors.
What type of attacks can be mounted against cookies?

Broadly speaking, we want to prevent attackers from *tampering* with our cookies (i.e. integrity) or *sniffing* their content (i.e. confidentiality).

First and foremost, transmitting cookies over an insecure connection (i.e. HTTP instead of HTTPS) exposes us to man in the middle attacks - the request sent to the server by the browser can be intercepted, read and its content modified arbitrarily.
The first line of defense is our API - it should reject requests sent over unencrypted channels. We can benefit from an additional layer of defense by marking newly created cookies as `Secure`: this instructs browsers to only attach the cookie to requests transmitted over secure connections.

The second major threat to the confidentiality and integrity of our cookies is JavaScript: scripts running client-side can interact with the cookie store, read/modify existing cookies or set new ones. As a rule of thumb, a least-privilege policy is a good default: cookies should not be visible to scripts unless there is a compelling reason to do otherwise. We can mark newly created cookies as `Http-Only` to hide them from client-side code - the browser will store them and attach them to outgoing requests, as usual, but scripts will not be able to see them.
`Http-Only` is a good default, but it is not a panacea - JavaScript code might not be able to access our `Http-Only` cookie, but there are ways to overwrite them[20] and trick the backend to perform some unexpected or undesired actions.
Last but not least, users can be a threat as well! They can freely manipulate the content of their cookie storage using the developer tools provided by their browser. While this might not be an issue when looking at flash messages, it definitely becomes a concern when working with other types of cookies (e.g. auth sessions, which we will be looking at shortly).

We should have multiple layers of defense.
We already know of an approach to ensure integrity, no matter what happens in the front-channel, don't we?
Message authentication codes (MAC), the ones we used to secure our query parameters! A cookie value with an HMAC tag attached is often referred to as a **signed cookie**. By verifying the tag on the backend we can be confident that the value of a signed cookie has not been tampered with, just like we did for query parameters.

10.6.4.15 `actix-web-flash-messages`

We could use the cookie API provided by `actix-web` to harden our cookie-based implementation of flash messages - some things are straight-forward (`Secure`, `Http-Only`), others requires a bit

[20] An attack known as "cookie jar overflow" can be used to delete pre-existing `Http-Only` cookies. The cookies can then be overwritten with a value set by the malicious script.

more work (HMAC), but they are all quite achievable if we put in some effort.
We have already covered HMAC tags in depth when discussing query parameters, so there would be little educational benefit in implementing signed cookies from scratch. We will instead plug in one of the crates from `actix-web`'s community ecosystem: `actix-web-flash-messages`[21].

`actix-web-flash-messages` provides a framework to work with flash messages in `actix-web`, closely modeled after Django's message framework.

Let's add it as a dependency:

```toml
#! Cargo.toml
# [...]
[dependencies]
actix-web-flash-messages = { version = "0.3", features = ["cookies"] }
# [...]
```

To start playing around with flash messages we need to register `FlashMessagesFramework` as a middleware on our `actix_web`'s `App`:

```rust
//! src/startup.rs
// [...]
use actix_web_flash_messages::FlashMessagesFramework;

fn run(/* */) -> Result<Server, std::io::Error> {
    // [...]
    let message_framework = FlashMessagesFramework::builder(todo!()).build();
    let server = HttpServer::new(move || {
        App::new()
            .wrap(message_framework.clone())
            .wrap(TracingLogger::default())
        // [...]
    })
    // [...]
}
```

`FlashMessagesFramework::builder` expects a *storage backend* as argument - where should flash messages be stored and retrieved from?
`actix-web-flash-messages` provides a cookie-based implementation, `CookieMessageStore`.

```rust
//! src/startup.rs
// [...]
use actix_web_flash_messages::storage::CookieMessageStore;

fn run(/* */) -> Result<Server, std::io::Error> {
    // [...]
    let message_store = CookieMessageStore::builder(todo!()).build();
    let message_framework = FlashMessagesFramework::builder(message_store).build();
    // [...]
}
```

`CookieMessageStore` enforces that the cookie used as storage is signed, therefore we must provide a `Key` to its builder. We can reuse the `hmac_secret` we introduced when working on HMAC tags for query parameters:

[21] Full disclosure: I am the author of `actix-web-flash-messages`.

```rust
//! src/startup.rs
// [...]
use secrecy::ExposeSecret;
use actix_web::cookie::Key;

fn run(/* */) -> Result<Server, std::io::Error> {
    // [...]
    let message_store = CookieMessageStore::builder(
        Key::from(hmac_secret.expose_secret().as_bytes())
    ).build();
    // [...]
}
```

We can now start to send `FlashMessages`.
Each `FlashMessage` has a level and a string of content. The message level can be used for filtering and rendering - for example:

- Only show flash messages at `info` level or above in a production environment, while retaining `debug` level messages for local development;
- Use different colours, in the UI, to display messages (e.g. red for errors, orange for warnings, etc.).

We can rework `POST /login` to send a `FlashMessage`:

```rust
//! src/routes/login/post.rs
// [...]
use actix_web_flash_messages::FlashMessage;

#[tracing::instrument(/* */)]
pub async fn login(/* */) -> Result</* */> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(/* */) => { /* */ }
        Err(e) => {
            let e = /* */;
            FlashMessage::error(e.to_string()).send();
            let response = HttpResponse::SeeOther()
                // No cookies here now!
                .insert_header((LOCATION, "/login"))
                .finish();
            // [...]
        }
    }
}
```

The `FlashMessagesFramework` middleware takes care of all the heavy-lifting behind the scenes - creating the cookie, signing it, setting the right properties, etc.
We can also attach multiple flash messages to a single response - the framework takes care of how they should be combined and represented in the storage layer.

How does the receiving side work? How do we read incoming flash messages in `GET /login`?

We can use the `IncomingFlashMessages` extractor:

```rust
//! src/routes/login/get.rs
// [...]
```

```rust
use actix_web_flash_messages::{IncomingFlashMessages, Level};
use std::fmt::Write;

// No need to access the raw request anymore!
pub async fn login_form(flash_messages: IncomingFlashMessages) -> HttpResponse {
    let mut error_html = String::new();
    for m in flash_messages.iter().filter(|m| m.level() == Level::Error) {
        writeln!(error_html, "<p><i>{}</i></p>", m.content()).unwrap();
    }
    HttpResponse::Ok()
        // No more removal cookie!
        .content_type(ContentType::html())
        .body(format!(/* */))
}
```

The code needs to change a bit to accommodate the chance of having received multiple flash messages, but overall it is almost equivalent. In particular, we no longer have to deal with the cookie API, neither to retrieve incoming flash messages nor to make sure that they get erased after having been read - `actix-web-flash-messages` takes care of it. The validity of the cookie signature is verified in the background as well, before the request handler is invoked.

What about our tests?
They are failing:

```
---- login::an_error_flash_message_is_set_on_failure stdout ----
thread 'login::an_error_flash_message_is_set_on_failure' panicked at
'assertion failed: `(left == right)`
  left: `"Ik4JlkXTiTlc507ERzy2Ob4Xc4qXAPzJ7MiX6EB04c4%3D%5B%7B%2[...]"`,
 right: `"Authentication failed"`'
```

Our assertions are a bit too close to the implementation details - we should only verify that the rendered HTML contains (or does not contain) the expected error message. Let's amend the test code:

```rust
//! tests/api/login.rs
// [...]

#[tokio::test]
async fn an_error_flash_message_is_set_on_failure() {
    // Arrange
    // [...]
    // Act - Part 1 - Try to login
    // [...]
    // Assert
    // No longer asserting facts related to cookies
    assert_is_redirect_to(&response, "/login");

    // Act - Part 2 - Follow the redirect
    let html_page = app.get_login_html().await;
    assert!(html_page.contains("<p><i>Authentication failed</i></p>"));

    // Act - Part 3 - Reload the login page
    let html_page = app.get_login_html().await;
    assert!(!html_page.contains("<p><i>Authentication failed</i></p>"));
}
```

The test should pass now.

10.7 Sessions

We focused for a while on what should happen on a failed login attempt. Time to swap: what do we expect to see after a successful login?

Authentication is meant to restrict access to functionality that requires higher privileges - in our case, the capability to send out a new issue of the newsletter to the entire mailing list. We want to build an administration panel - we will have a `/admin/dashboard` page, restricted to logged-in users, to access all admin functionality.
We will get there in stages. As the very first milestone, we want to:

- redirect to `/admin/dashboard` after a successful login attempt to show a `Welcome <username>!` greeting message;
- if a user tries to navigate directly to `/admin/dashboard` and they are not logged in, they will be redirected to the login form.

This plan requires **sessions**.

10.7.1 Session-based Authentication

Session-based authentication is a strategy to avoid asking users to provide their password on every single page. Users are asked to authenticate once, via a login form: if successful, the server generates a one-time secret - an authenticated session token[22].
The backend API will accept the session token instead of the username/password combination and grant access to the restricted functionality. The session token must be provided on every request - this is why session tokens are stored as cookies. The browser will make sure to attach the cookie to all outgoing requests for the API.

From a security point of view, a valid session token is as powerful as the corresponding authentication secrets - e.g. the username/password combination, biometrics or physical second factors. We must take extreme care to avoid exposing session tokens to attackers.

OWASP provides extensive guidance on how to secure sessions - we will be implementing most of their recommendations in the next sections.

10.7.2 Session Store

Let's start to think about the implementation!
Based on what we discussed so far, we need the API to generate a session token after a successful login. The token value must be unpredictable - we do not want attackers to be able to generate or guess a valid session token[23]. OWASP recommends using a cryptographically secure pseudoran-

[22] Naming can be quite confusing - we are using the terms *session token*/*session cookie* to refer to the client-side cookie associated to a **user** session. Later in this chapter, we will talk about the lifecycle of cookies, where *session cookie* refers to a cookie whose lifetime is tied to a **browser** session. I'd love to change the naming to be clearer, but this ambiguity is now part of the industry terminology and there is no point in shielding you.

[23] A common example of poorly implemented sessions uses a monotonically increasing integer as session token - e.g. 6, 7, 8, etc. It is easy enough to "explore" nearby numbers by modifying the cookie stored in your browser until you manage to find another logged-in user - bingo, you are in! Not great.

dom number generator (CSPRNG).
Randomness on its own is not enough - we also need uniqueness. If we were to associate two users with the same session token we would be in trouble:

- we could be granting higher privileges to one of the two compared to what they deserve;
- we risk exposing personal or confidential information, such as names, emails, past activity, etc.

We need a **session store** - the server must remember the tokens it has generated in order to authorize future requests for logged-in users. We also want to associate information to each active session - this is known as **session state**.

10.7.3 Choosing A Session Store

During the lifecycle of a session we need to perform the following operations:

- **creation**, when a user logs in;
- **retrieval**, using the session tokens extracted from the cookie attached to the incoming requests;
- **update**, when a logged-in user performs some actions that lead to a change in their session state;
- **deletion**, when the user logs out.

These are commonly known as *CRUD* (**c**reate, **d**elete, **r**ead, **u**pdate).

We also need some form of **expiration** - sessions are meant to be short-lived. Without a clean-up mechanism we are going to end up using more space for outdated/stale session than active ones.

10.7.3.1 Postgres

Would Postgres be a viable session store?
We could create a new `sessions` table with the token as primary index - an easy way to ensure token uniqueness. We have a few options for the session state:

- "classical" relational modelling, using a normalised schema (i.e. the way we approached storage of our application state);
- a single `state` column holding a collection of key-value pairs, using the `jsonb` data type.

Unfortunately, there is no built-in mechanism for row expiration in Postgres. We would have to add a `expires_at` column and trigger a cleanup job on a regular schedule to purge stale sessions - somewhat cumbersome.

10.7.3.2 Redis

Redis is another popular option when it comes to session storage.
Redis is an in-memory database - it uses RAM instead of disk for storage, trading off durability for speed. It is great fit, in particular, for data that can be modelled as a collection of key-value pairs. It also provides native support for expiration - we can attach a time-to-live to all values and Redis will take care of disposal.

How would it work for sessions?
Our application never manipulates sessions in bulk - we always work on a single session at a time,

identified using its token. Therefore, we can use the session token as key while the value is the JSON representation of the session state - the application takes care of serialization/deserialization. Sessions are meant to be short-lived - no reason to be concerned by the usage of RAM instead of disk for persistence, the speed boost is a nice side effect!

As you might have guessed at this point, we will be using Redis as our session storage backend!

10.7.4 actix-session

actix-session provides session management for actix-web applications. Let's add it to our dependencies:

```
#! Cargo.toml
# [...]
[dependencies]
# [...]
actix-session = "0.6"
```

The key type in actix-session is SessionMiddleware - it that takes care of loading the session data, tracking changes to the state and persisting them at the end of the request/response lifecycle. To build an instance of SessionMiddleware we need to provide a storage backend and a secret key to sign (or encrypt) the session cookie. The approach is quite similar to the one used by FlashMessagesFramework in actix-web-flash-messages.

```rust
//! src/startup.rs
// [...]
use actix_session::SessionMiddleware;

fn run(
    // [...]
) -> Result<Server, std::io::Error> {
    // [...]
    let secret_key = Key::from(hmac_secret.expose_secret().as_bytes());
    let message_store = CookieMessageStore::builder(secret_key.clone()).build();
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            .wrap(message_framework.clone())
            .wrap(SessionMiddleware::new(todo!(), secret_key.clone()))
            .wrap(TracingLogger::default())
        // [...]
    })
    // [...]
}
```

actix-session is quite flexible when it comes to storage - you can provide your own by implementing the SessionStore trait. It also offers some implementations out of the box, hidden behind a set of feature flags - including a Redis backend. Let's enable it:

```
#! Cargo.toml
# [...]
[dependencies]
# [...]
actix-session = { version = "0.6", features = ["redis-rs-tls-session"] }
```

We can now access `RedisSessionStore`. To build one we will have to pass a Redis connection string as input - let's add `redis_uri` to our configuration struct:

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

#[derive(serde::Deserialize, Clone)]
pub struct Settings {
    // [...]
    // We have not created a stand-alone settings struct for Redis,
    // let's see if we need more than the uri first!
    // The URI is marked as secret because it may embed a password.
    pub redis_uri: Secret<String>,
}
```

```yaml
# configuration/base.yaml
# 6379 is Redis' default port
redis_uri: "redis://127.0.0.1:6379"
# [...]
```

Let's use it to build a `RedisSessionStore` instance:

```rust
//! src/startup.rs
// [...]
use actix_session::storage::RedisSessionStore;

impl Application {
    // Async now! We also return anyhow::Error instead of std::io::Error
    pub async fn build(configuration: Settings) -> Result<Self, anyhow::Error> {
        // [...]
        let server = run(
            // [...]
            configuration.redis_uri
        ).await?;
        // [...]
    }
}

// Now it's asynchronous!
async fn run(
    // [...]
    redis_uri: Secret<String>,
// Returning anyhow::Error instead of std::io::Error
) -> Result<Server, anyhow::Error> {
    // [...]
    let redis_store = RedisSessionStore::new(redis_uri.expose_secret()).await?;
    let server = HttpServer::new(move || {
        App::new()
            .wrap(message_framework.clone())
            .wrap(SessionMiddleware::new(redis_store.clone(), secret_key.clone()))
            .wrap(TracingLogger::default())
            // [...]
        })
    // [...]
}
```

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

#[tokio::main]
// anyhow::Result now instead of std::io::Error
async fn main() -> anyhow::Result<()> {
    // [...]
}
```

Time to add a running Redis instance to our setup.

10.7.4.1 Redis In Our Development Setup

We need to run a Redis container alongside the Postgres container in our CI pipeline - check out the updated YAML in the book repository.

We also need a running Redis container on our development machine to execute the test suite and launch the application. Let's add a script to launch it:

```bash
# scripts/init_redis.sh
#!/usr/bin/env bash
set -x
set -eo pipefail

# if a redis container is running, print instructions to kill it and exit
RUNNING_CONTAINER=$(docker ps --filter 'name=redis' --format '{{.ID}}')
if [[ -n $RUNNING_CONTAINER ]]; then
  echo >&2 "there is a redis container already running, kill it with"
  echo >&2 "    docker kill ${RUNNING_CONTAINER}"
  exit 1
fi

# Launch Redis using Docker
docker run \
    -p "6379:6379" \
    -d \
    --name "redis_$(date '+%s')" \
    redis:6

>&2 echo "Redis is ready to go!"
```

The script needs to be marked as executable and then launched:

```bash
chmod +x ./scripts/init_redis.sh
./script/init_redis.sh
```

10.7.4.2 Redis On Digital Ocean

Digital Ocean does not support the creation of a development Redis cluster via the `spec.yaml` file. You need to go through their dashboard - create a new Redis cluster here. Make sure to select the datacenter where you deployed the application. Once the cluster has been created, you have to go through a quick "Get started" flow to configure a few knobs (trusted sources, eviction policy, etc.).

At the end of the "Get started" flow you will be able to copy a connection string to your newly provisioned Redis instance. The connection string embeds a username and a password, therefore we must treat it as a secret. We will inject its value into the application using an environment value - set `APP_REDIS_URI` from the `Settings` panel in your application console.

10.7.5 Admin Dashboard

Our session store is now up and running in all the environments we care about. It is time to actually do something with it!
Let's create the skeleton for a new page, the admin dashboard.

```rust
//! src/routes/admin/mod.rs
mod dashboard;

pub use dashboard::admin_dashboard;
```

```rust
//! src/routes/admin/dashboard.rs
use actix_web::HttpResponse;

pub async fn admin_dashboard() -> HttpResponse {
    HttpResponse::Ok().finish()
}
```

```rust
//! src/routes/mod.rs
// [...]
mod admin;
pub use admin::*;
```

```rust
//! src/startup.rs
use crate::routes::admin_dashboard;
// [...]
async fn run(/* */) -> Result<Server, anyhow::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/admin/dashboard", web::get().to(admin_dashboard))
            // [...]
    })
    // [...]
}
```

10.7.5.1 Redirect On Login Success

Let's start to work on the first milestone:

> Redirect to `/admin/dashboard` after a successful login attempt to show a `Welcome <username>!` greeting message;

We can encode the requirements in an integration test:

```rust
//! tests/api/login.rs
// [...]

#[tokio::test]
async fn redirect_to_admin_dashboard_after_login_success() {
    // Arrange
    let app = spawn_app().await;

    // Act - Part 1 - Login
    let login_body = serde_json::json!({
        "username": &app.test_user.username,
        "password": &app.test_user.password
    });
    let response = app.post_login(&login_body).await;
    assert_is_redirect_to(&response, "/admin/dashboard");

    // Act - Part 2 - Follow the redirect
    let html_page = app.get_admin_dashboard().await;
    assert!(html_page.contains(&format!("Welcome {}", app.test_user.username)));
}
```

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]
    pub async fn get_admin_dashboard(&self) -> String {
        self.api_client
            .get(&format!("{}/admin/dashboard", &self.address))
            .send()
            .await
            .expect("Failed to execute request.")
            .text()
            .await
            .unwrap()
    }
}
```

The test should fail:

```
---- login::redirect_to_admin_dashboard_after_login_success stdout ----
thread 'login::redirect_to_admin_dashboard_after_login_success' panicked at
'assertion failed: `(left == right)`
  left: `"/"`,
 right: `"/admin/dashboard"`'
```

Getting past the first assertion is easy enough - we just need to change the `Location` header in the response returned by `POST /login`:

```rust
//! src/routes/login/post.rs
// [...]

#[tracing::instrument(/* */)]
pub async fn login(/* */) -> Result</* */> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(/* */) => {
            // [...]
            Ok(HttpResponse::SeeOther()
                .insert_header((LOCATION, "/admin/dashboard"))
                .finish())
        }
        // [...]
    }
}
```

The test will now fail on the second assertion:

```
---- login::redirect_to_admin_dashboard_after_login_success stdout ----
thread 'login::redirect_to_admin_dashboard_after_login_success' panicked at
'assertion failed: html_page.contains(...)',
```

Time to put those sessions to work.

10.7.5.2 `Session`

We need to identify the user once it lands on `GET /admin/dashboard` after following the redirect returned by `POST /login` - this is a perfect usecase for sessions.
We will store the user identifier into the session state in `login` and then retrieve it from the session state in `admin_dashboard`.

We need to become familiar with `Session`, the second key type from `actix_session`.
`SessionMiddleware` does all the heavy lifting of checking for a session cookie in incoming requests - if it finds one, it loads the corresponding session state from the chosen storage backend. Otherwise, it creates a new empty session state.
We can then use `Session` as an extractor to interact with that state in our request handlers.

Let's see it in action in `POST /login`:

```rust
//! src/routes/login/post.rs
use actix_session::Session;
// [...]

#[tracing::instrument(
    skip(form, pool, session),
    // [...]
)]
pub async fn login(
    // [...]
    session: Session,
) -> Result</* */> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(user_id) => {
            // [...]
            session.insert("user_id", user_id);
            Ok(HttpResponse::SeeOther()
                .insert_header((LOCATION, "/admin/dashboard"))
                .finish())
        }
        // [...]
    }
}
```

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

```toml
# We need to add the `serde` feature
uuid = { version = "0.8.1", features = ["v4", "serde"] }
```

You can think of `Session` as a handle on a `HashMap` - you can insert and retrieve values against `String` keys.
The values you pass in must be serializable - `actix-session` converts them into JSON behind the scenes. That's why we had to add the `serde` feature to our `uuid` dependency.

Serialisation implies the possibility of failure - if you run `cargo check` you will see that the compiler warns us that we are not handling the `Result` returned by `session.insert`. Let's take care of that:

```rust
//! src/routes/login/post.rs
// [...]
#[tracing::instrument(/* */)]
pub async fn login(/* */) -> Result<HttpResponse, InternalError<LoginError>> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(user_id) => {
            // [...]
            session
                .insert("user_id", user_id)
                .map_err(|e| login_redirect(LoginError::UnexpectedError(e.into())))?;
            // [...]
        }
        Err(e) => {
            let e = match e {
                AuthError::InvalidCredentials(_) => LoginError::AuthError(e.into()),
                AuthError::UnexpectedError(_) => LoginError::UnexpectedError(e.into()),
            };
            Err(login_redirect(e))
        }
    }
}

// Redirect to the login page with an error message.
fn login_redirect(e: LoginError) -> InternalError<LoginError> {
    FlashMessage::error(e.to_string()).send();
    let response = HttpResponse::SeeOther()
        .insert_header((LOCATION, "/login"))
        .finish();
    InternalError::from_response(e, response)
}
```

If something goes wrong, the user will be redirected back to the `/login` page with an appropriate error message.

What does `Session::insert` actually do, though?
All operations performed against `Session` are executed in memory - they do not affect the state of the session as seen by the storage backend. After the handler returns a response, `SessionMiddleware` will inspect the in-memory state of `Session` - if it changed, it will call Redis to update (or create) the state. It will also take care of setting a session cookie on the client, if there wasn't one already.

Does it work though? Let's try to get the `user_id` on the other side!

```rust
//! src/routes/admin/dashboard.rs
use actix_session::Session;
use actix_web::{web, HttpResponse};
use uuid::Uuid;

// Return an opaque 500 while preserving the error's root cause for logging.
fn e500<T>(e: T) -> actix_web::Error
where
    T: std::fmt::Debug + std::fmt::Display + 'static
{
    actix_web::error::ErrorInternalServerError(e)
}

pub async fn admin_dashboard(
    session: Session
) -> Result<HttpResponse, actix_web::Error> {
    let _username = if let Some(user_id) = session
        .get::<Uuid>("user_id")
        .map_err(e500)?
    {
        todo!()
    } else {
        todo!()
    };
    Ok(HttpResponse::Ok().finish())
}
```

When using `Session::get` we must specify what type we want to deserialize the session state entry into - a `Uuid` in our case. Deserialization may fail, so we must handle the error case.
Now that we have the `user_id`, we can use it to fetch the username and return the "Welcome {username}!" message we talked about before.

```rust
//! src/routes/admin/dashboard.rs
// [...]
use actix_web::http::header::ContentType;
use actix_web::web;
use anyhow::Context;
use sqlx::PgPool;

pub async fn admin_dashboard(
    session: Session,
    pool: web::Data<PgPool>,
) -> Result<HttpResponse, actix_web::Error> {
    let username = if let Some(user_id) = session
        .get::<Uuid>("user_id")
        .map_err(e500)?
    {
        get_username(user_id, &pool).await.map_err(e500)?
    } else {
        todo!()
    };
    Ok(HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(
            r#"<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Admin dashboard</title>
</head>
<body>
    <p>Welcome {username}!</p>
</body>
</html>"#
        )))
}

#[tracing::instrument(name = "Get username", skip(pool))]
async fn get_username(
    user_id: Uuid,
    pool: &PgPool
) -> Result<String, anyhow::Error> {
    let row = sqlx::query!(
        r#"
        SELECT username
        FROM users
        WHERE user_id = $1
        "#,
        user_id,
    )
    .fetch_one(pool)
    .await
    .context("Failed to perform a query to retrieve a username.")?;
    Ok(row.username)
}
```

Our integration test should pass now!

Stay there though, we are not finished yet - as it stands, our login flow is potentially vulnerable to session fixation attacks.
Sessions can be used for more than authentication - e.g. to keep track of what items have been added to the basket when shopping in "guest" mode. This implies that a user might be associated to an *anonymous* session and, after they authenticate, to a *privileged* session. This can be leveraged by attackers.
Websites go to great lengths to prevent malicious actors from sniffing session tokens, leading to another attack strategy - seed the user's browser with a **known** session token **before** they log in, wait for authentication to happen and, boom, you are in!

There is a simple countermeasure we can take to disrupt this attack - rotating the session token when the user logs in.
This is such a common practice that you will find it supported in the session management API of all major web frameworks - including `actix-session`, via `Session::renew`. Let's add it in:

```
//! src/routes/login/post.rs
// [...]
#[tracing::instrument(/* */)]
pub async fn login(/* */) -> Result<HttpResponse, InternalError<LoginError>> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(user_id) => {
            // [...]
            session.renew();
            session
                .insert("user_id", user_id)
                .map_err(|e| login_redirect(LoginError::UnexpectedError(e.into())))?;
            // [...]
        }
        // [...]
    }
}
```

Now we can sleep better.

10.7.5.3 A Typed Interface To `Session`

`Session` is powerful but, taken as is, it is a brittle foundation to build your application state-handling on. We are accessing data using a string-based API, being careful to use the same keys and types on both sides - insertion and retrieval. It works when the state is very simple, but it quickly degrades into a mess if you have several routes accessing the same data - how can you be sure that you updated all of them when you want to evolve the schema? How do we prevent a key typo from causing a production outage?

Tests can help, but we can use the type system to make the problem go away entirely. We will build a strongly-typed API on top of `Session` to access and modify the state - no more string keys and type casting in our request handlers.
`Session` is a foreign type (defined in `actix-session`) therefore we must use the extension trait pattern:

```rust
//! src/lib.rs
// [...]
pub mod session_state;
```

```rust
//! src/session_state.rs
use actix_session::Session;
use uuid::Uuid;

pub struct TypedSession(Session);

impl TypedSession {
    const USER_ID_KEY: &'static str = "user_id";

    pub fn renew(&self) {
        self.0.renew();
    }

    pub fn insert_user_id(&self, user_id: Uuid) -> Result<(), serde_json::Error> {
        self.0.insert(Self::USER_ID_KEY, user_id)
    }

    pub fn get_user_id(&self) -> Result<Option<Uuid>, serde_json::Error> {
        self.0.get(Self::USER_ID_KEY)
    }
}
```

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

```toml
[dependencies]
serde_json = "1"
# [...]
```

How will the request handlers build an instance of `TypedSession`?
We could provide a constructor that takes a `Session` as argument. Another option is to make `TypedSession` itself an `actix-web` extractor - let's try that out!

```rust
//! src/session_state.rs
// [...]
use actix_session::SessionExt;
use actix_web::dev::Payload;
use actix_web::{FromRequest, HttpRequest};
use std::future::{Ready, ready};

impl FromRequest for TypedSession {
    // This is a complicated way of saying
    // "We return the same error returned by the
    // implementation of `FromRequest` for `Session`".
    type Error = <Session as FromRequest>::Error;
    // Rust does not yet support the `async` syntax in traits.
    // From request expects a `Future` as return type to allow for extractors
    // that need to perform asynchronous operations (e.g. a HTTP call)
    // We do not have a `Future`, because we don't perform any I/O,
    // so we wrap `TypedSession` into `Ready` to convert it into a `Future` that
    // resolves to the wrapped value the first time it's polled by the executor.
    type Future = Ready<Result<TypedSession, Self::Error>>;

    fn from_request(req: &HttpRequest, _payload: &mut Payload) -> Self::Future {
        ready(Ok(TypedSession(req.get_session())))
    }
}
```

It is just three lines long, but it does probably expose you to a few new Rust concepts/constructs. Take the time you need to go line by line and properly understand what is happening - or, if you prefer, understand the gist and come back later to deep dive!

We can now swap `Session` for `TypedSession` in our request handlers:

```rust
//! src/routes/login/post.rs
// You can now remove the `Session` import
use crate::session_state::TypedSession;
// [...]

#[tracing::instrument(/* */)]
pub async fn login(
    // [...]
    // Changed from `Session` to `TypedSession`!
    session: TypedSession,
) -> Result</* */> {
    // [...]
    match validate_credentials(/* */).await {
        Ok(user_id) => {
            // [...]
            session.renew();
            session
                .insert_user_id(user_id)
                .map_err(|e| login_redirect(LoginError::UnexpectedError(e.into())))?;
            // [...]
        }
        // [...]
    }
}
```

```rust
//! src/routes/admin/dashboard.rs
// You can now remove the `Session` import
use crate::session_state::TypedSession;
// [...]

pub async fn admin_dashboard(
    // Changed from `Session` to `TypedSession`!
    session: TypedSession,
    // [...]
) -> Result</* */> {
    let username = if let Some(user_id) = session.get_user_id().map_err(e500)? {
        // [...]
    } else {
        todo!()
    };
    // [...]
}
```

The test suite should stay green.

10.7.5.4 Reject Unauthenticated Users

We can now take care of the second milestone:

> If a user tries to navigate directly to `/admin/dashboard` and they are not logged in, they will be redirected to the login form.

Let's encode the requirements in an integration test, as usual:

```rust
//! tests/api/admin_dashboard.rs
use crate::helpers::{spawn_app, assert_is_redirect_to};

#[tokio::test]
async fn you_must_be_logged_in_to_access_the_admin_dashboard() {
    // Arrange
    let app = spawn_app().await;

    // Act
    let response = app.get_admin_dashboard_html().await;

    // Assert
    assert_is_redirect_to(&response, "/login");
}
```

```rust
//! tests/api/helpers.rs
//!
impl TestApp {
    // [...]
    pub async fn get_admin_dashboard(&self) -> reqwest::Response {
        self.api_client
            .get(&format!("{}/admin/dashboard", &self.address))
            .send()
            .await
            .expect("Failed to execute request.")
    }

    pub async fn get_admin_dashboard_html(&self) -> String {
        self.get_admin_dashboard().await.text().await.unwrap()
    }
}
```

The test should fail - the handler panics.
We can fix it by fleshing out that `todo!()`:

```rust
//! src/routes/admin/dashboard.rs
use actix_web::http::header::LOCATION;
// [...]

pub async fn admin_dashboard(
    session: TypedSession,
    pool: web::Data<PgPool>,
) -> Result<HttpResponse, actix_web::Error> {
    let username = if let Some(user_id) = session.get_user_id().map_err(e500)? {
        // [...]
    } else {
        return Ok(HttpResponse::SeeOther()
            .insert_header((LOCATION, "/login"))
            .finish());
    };
    // [...]
}
```

The test will pass now.

10.8 Seed Users

Everything looks great - in our test suite.
We have not done any exploratory testing for the most recent functionality - we stopped messing around in the browser more or less at the same time we started to work on the happy path. It is not a coincidence - we currently **cannot** exercise the happy path!
There is no user in the database and we do not have a sign up flow for admins - the implicit expectation has been that the application owner would become the first admin of the newsletter *somehow*![24]

It is time to make that a reality.
We will create a seed user - i.e. add a migration that creates a user into a database when the application is deployed for the first time. The seed user will have a pre-determined username and password[25]; they will then be able to change their password after they log in for the first time.

[24] The seed admin should then be able to invite more collaborators if they wish to do so. You could implement this login-protected functionality as an exercise! Look at the subscription flow for inspiration.

[25] In a more advanced scenario, the username and the password of the seed user could be configured by the application operator when they trigger the first deployment of the newsletter - e.g. they could be prompted to provide both

10.8.1 Database Migration

Let's create a new migration using `sqlx`:

```
sqlx migrate add seed_user
```

We need to insert a new row into the `users` table. We need:

- a user id (UUID);
- a username;
- a PHC string.

Pick your favourite UUID generator to get a valid user id. We will use `admin` as username.
Getting a PHC string is a bit more cumbersome - we will use `everythinghastostartsomewhere` as a password, but how do we generate the corresponding PHC string?
We can cheat by leveraging the code we wrote in our test suite:

```rust
//! tests/api/helpers.rs
// [...]

impl TestUser {
    pub fn generate() -> Self {
        Self {
            // [...]
            // password: Uuid::new_v4().to_string(),
            password: "everythinghastostartsomewhere".into(),
        }
    }

    async fn store(&self, pool: &PgPool) {
        // [...]
        let password_hash = /* */;
        // `dbg!` is a macro that prints and returns the value
        // of an expression for quick and dirty debugging.
        dbg!(&password_hash);
        // [...]
    }
}
```

This is just a temporary edit - it is then enough to run `cargo test -- --nocapture` to get a well-formed PHC string for our migration script. Revert the changes once you have it.

The migration script will look like this:

```sql
--- 20211217223217_seed_user.sql
INSERT INTO users (user_id, username, password_hash)
VALUES (
    'ddf8994f-d522-4659-8d02-c1d479057be6',
    'admin',
    '$argon2id$v=19$m=15000,t=2,p=1$OEx/rcq+3ts//WUDzGNl2g$Am8UFBA4w5NJEmAtquGvBmAlu92q/VQcaoL5AyJPfc8'
);
```

```
sqlx migrate run
```

Run the migration and then launch your application with `cargo run` - you should finally be able to log in successfully!

by a command-line application used to provide a streamlined installation process.

If everything works as expected, a "Welcome admin!" message should greet you at `/admin/dashboard`. Congrats!

10.8.2 Password Reset

Let's look at the current situation from another perspective - we just provisioned a highly privileged user with a known username/password combination. This is dangerous territory.
We need to give our seed user the possibility to change their password. It is going to be the first piece of functionality hosted on the admin dashboard!

No new concepts will be required to build this functionality - take this section as an opportunity to revise and make sure that you have a solid grasp on everything we covered so far!

10.8.2.1 Form Skeleton

Let's start by putting in place the required scaffolding. It is a form-based flow, just like the login one - we need a `GET` endpoint to return the HTML form and a `POST` endpoint to process the submitted information:

```rust
//! src/routes/admin/mod.rs
// [...]
mod password;
pub use password::*;
```

```rust
//! src/routes/admin/password/mod.rs
mod get;
pub use get::change_password_form;
mod post;
pub use post::change_password;
```

```rust
//! src/routes/admin/password/get.rs
use actix_web::http::header::ContentType;
use actix_web::HttpResponse;

pub async fn change_password_form() -> Result<HttpResponse, actix_web::Error> {
    Ok(HttpResponse::Ok().content_type(ContentType::html()).body(
        r#"<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Change Password</title>
</head>
<body>
    <form action="/admin/password" method="post">
        <label>Current password
            <input
                type="password"
                placeholder="Enter current password"
                name="current_password"
            >
        </label>
        <br>
        <label>New password
            <input
                type="password"
                placeholder="Enter new password"
                name="new_password"
            >
        </label>
        <br>
        <label>Confirm new password
            <input
                type="password"
                placeholder="Type the new password again"
                name="new_password_check"
            >
        </label>
        <br>
        <button type="submit">Change password</button>
    </form>
    <p><a href="/admin/dashboard">&lt;- Back</a></p>
</body>
</html>"#,
    ))
}
```

```
//! src/routes/admin/password/post.rs
use actix_web::{HttpResponse, web};
use secrecy::Secret;

#[derive(serde::Deserialize)]
pub struct FormData {
    current_password: Secret<String>,
    new_password: Secret<String>,
    new_password_check: Secret<String>,
}

pub async fn change_password(
    form: web::Form<FormData>,
) -> Result<HttpResponse, actix_web::Error> {
    todo!()
}
```

```
//! src/startup.rs
use crate::routes::{change_password, change_password_form};
// [...]

async fn run(/* */) -> Result<Server, anyhow::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/admin/password", web::get().to(change_password_form))
            .route("/admin/password", web::post().to(change_password))
            // [...]
    })
    // [...]
}
```

Just like the admin dashboard itself, we do not want to show the change password form to users who are not logged in. Let’s add two integration tests:

```rust
//! tests/api/main.rs
mod change_password;
// [...]
```

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]
    pub async fn get_change_password(&self) -> reqwest::Response {
        self.api_client
            .get(&format!("{}/admin/password", &self.address))
            .send()
            .await
            .expect("Failed to execute request.")
    }

    pub async fn post_change_password<Body>(&self, body: &Body) -> reqwest::Response
        where
            Body: serde::Serialize,
    {
        self.api_client
            .post(&format!("{}/admin/password", &self.address))
            .form(body)
            .send()
            .await
            .expect("Failed to execute request.")
    }
}
```

```rust
//! tests/api/change_password.rs
use crate::helpers::{spawn_app, assert_is_redirect_to};
use uuid::Uuid;

#[tokio::test]
async fn you_must_be_logged_in_to_see_the_change_password_form() {
    // Arrange
    let app = spawn_app().await;

    // Act
    let response = app.get_change_password().await;

    // Assert
    assert_is_redirect_to(&response, "/login");
}

#[tokio::test]
async fn you_must_be_logged_in_to_change_your_password() {
    // Arrange
    let app = spawn_app().await;
    let new_password = Uuid::new_v4().to_string();

    // Act
    let response = app
        .post_change_password(&serde_json::json!({
            "current_password": Uuid::new_v4().to_string(),
            "new_password": &new_password,
            "new_password_check": &new_password,
        }))
        .await;

    // Assert
    assert_is_redirect_to(&response, "/login");
}
```

We can then satisfy the requirements by adding a check in the request handlers[26]:

```rust
//! src/routes/admin/password/get.rs
use crate::session_state::TypedSession;
use crate::utils::{e500, see_other};
// [...]

pub async fn change_password_form(
    session: TypedSession
) -> Result</* */> {
    if session.get_user_id().map_err(e500)?.is_none() {
        return Ok(see_other("/login"));
    };
    // [...]
}
```

```rust
//! src/routes/admin/password/post.rs
use crate::session_state::TypedSession;
use crate::utils::{e500, see_other};
// [...]

pub async fn change_password(
    // [...]
    session: TypedSession,
) -> Result<HttpResponse, actix_web::Error> {
    if session.get_user_id().map_err(e500)?.is_none() {
        return Ok(see_other("/login"));
    };
    todo!()
}
```

```rust
//! src/utils.rs
use actix_web::HttpResponse;
use actix_web::http::header::LOCATION;

// Return an opaque 500 while preserving the error root's cause for logging.
pub fn e500<T>(e: T) -> actix_web::Error
where
    T: std::fmt::Debug + std::fmt::Display + 'static,
{
    actix_web::error::ErrorInternalServerError(e)
}

pub fn see_other(location: &str) -> HttpResponse {
```

[26] An alternative approach, to spare us the repetition, is to create a middleware that wraps all the endpoints nested under the `/admin/` prefix. The middleware checks the session state and redirects the visitor to `/login` if they are not logged in. If you like a challenge, give it a try! Beware though: `actix-web`'s middlewares can be tricky to implement due to the lack of async syntax in traits.

```rust
    HttpResponse::SeeOther()
        .insert_header((LOCATION, location))
        .finish()
}
```

```rust
//! src/lib.rs
// [...]
pub mod utils;
```

```rust
//! src/routes/admin/dashboard.rs
// The definition of e500 has been moved to src/utils.rs
use crate::utils::e500;
// [...]
```

We do not want the change password form to be an orphan page either - let's add a list of available actions to our admin dashboard, with a link to our new page:

```rust
//! src/routes/admin/dashboard.rs
// [...]

pub async fn admin_dashboard(/* */) -> Result</* */> {
    // [...]
    Ok(HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(
            r#"<!DOCTYPE html>
<html lang="en">
<head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8">
    <title>Admin dashboard</title>
</head>
<body>
    <p>Welcome {username}!</p>
    <p>Available actions:</p>
    <ol>
        <li><a href="/admin/password">Change password</a></li>
    </ol>
</body>
</html>"#,
        )))
}
```

10.8.2.2 Unhappy Path: New Passwords Do Not Match

We have taken care of all the preliminary steps, it is time to start working on the core functionality. Let's start with an unhappy case - we asked the user to write the new password twice and the two entries do not match. We expect to be redirected back to the form with an appropriate error message.

```rust
//! tests/api/change_password.rs
// [...]

#[tokio::test]
async fn new_password_fields_must_match() {
    // Arrange
    let app = spawn_app().await;
    let new_password = Uuid::new_v4().to_string();
    let another_new_password = Uuid::new_v4().to_string();

    // Act - Part 1 - Login
    app.post_login(&serde_json::json!({
        "username": &app.test_user.username,
        "password": &app.test_user.password
    }))
    .await;

    // Act - Part 2 - Try to change password
    let response = app
        .post_change_password(&serde_json::json!({
            "current_password": &app.test_user.password,
            "new_password": &new_password,
            "new_password_check": &another_new_password,
        }))
        .await;
    assert_is_redirect_to(&response, "/admin/password");

    // Act - Part 3 - Follow the redirect
    let html_page = app.get_change_password_html().await;
    assert!(html_page.contains(
        "<p><i>You entered two different new passwords - \
         the field values must match.</i></p>"
    ));
}
```

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]

    pub async fn get_change_password_html(&self) -> String {
        self.get_change_password().await.text().await.unwrap()
    }
}
```

The test fails because the request handler panics. Let's fix it:

```rust
//! src/routes/admin/password/post.rs
use secrecy::ExposeSecret;
// [...]

pub async fn change_password(/* */) -> Result</* */> {
    // [...]
    // `Secret<String>` does not implement `Eq`,
    // therefore we need to compare the underlying `String`.
    if form.new_password.expose_secret() != form.new_password_check.expose_secret() {
        return Ok(see_other("/admin/password"));
    }
    todo!()
}
```

That takes care of the redirect, the first part of the test, but it does not handle the error message:

```
---- change_password::new_password_fields_must_match stdout ----
thread 'change_password::new_password_fields_must_match' panicked at
'assertion failed: html_page.contains(...)',
```

We have gone through this journey before for the login form - we can use a flash message again!

```rust
//! src/routes/admin/password/post.rs
// [...]
use actix_web_flash_messages::FlashMessage;

pub async fn change_password(/* */) -> Result</* */> {
    // [...]
    if form.new_password.expose_secret() != form.new_password_check.expose_secret() {
        FlashMessage::error(
            "You entered two different new passwords - the field values must match.",
        )
        .send();
        // [...]
    }
    todo!()
}
```

```rust
//! src/routes/admin/password/get.rs
// [...]
use actix_web_flash_messages::IncomingFlashMessages;
use std::fmt::Write;

pub async fn change_password_form(
    session: TypedSession,
    flash_messages: IncomingFlashMessages,
) -> Result<HttpResponse, actix_web::Error> {
    // [...]

    let mut msg_html = String::new();
    for m in flash_messages.iter() {
        writeln!(msg_html, "<p><i>{}</i></p>", m.content()).unwrap();
    }

    Ok(HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(
            r#"<!-- [...] -->
<body>
    {msg_html}
    <!-- [...] -->
</body>
</html>"#,
        )))
}
```

The test should pass.

10.8.2.3 Unhappy Path: The Current Password Is Invalid

You might have noticed that we require the user to provide its current password as part of the form. This is to prevent an attacker who managed to acquire a valid session token from locking the legitimate user out of their account.

Let's add an integration test to specify what we expect to see when the provided current password is invalid:

```rust
//! tests/api/change_password.rs
// [...]

#[tokio::test]
async fn current_password_must_be_valid() {
    // Arrange
    let app = spawn_app().await;
    let new_password = Uuid::new_v4().to_string();
    let wrong_password = Uuid::new_v4().to_string();

    // Act - Part 1 - Login
    app.post_login(&serde_json::json!({
        "username": &app.test_user.username,
        "password": &app.test_user.password
    }))
    .await;

    // Act - Part 2 - Try to change password
    let response = app
        .post_change_password(&serde_json::json!({
            "current_password": &wrong_password,
            "new_password": &new_password,
            "new_password_check": &new_password,
        }))
        .await;

    // Assert
    assert_is_redirect_to(&response, "/admin/password");

    // Act - Part 3 - Follow the redirect
    let html_page = app.get_change_password_html().await;
    assert!(html_page.contains(
        "<p><i>The current password is incorrect.</i></p>"
    ));
}
```

To validate the value passed as `current_password` we need to retrieve the username and then invoke the `validate_credentials` routine, the one powering our login form.
Let's start with the username:

```rust
//! src/routes/admin/password/post.rs
use crate::routes::admin::dashboard::get_username;
use sqlx::PgPool;
// [...]

pub async fn change_password(
    // [...]
    pool: web::Data<PgPool>,
) -> Result<HttpResponse, actix_web::Error> {
    let user_id = session.get_user_id().map_err(e500)?;
    if user_id.is_none() {
        return Ok(see_other("/login"));
    };
    let user_id = user_id.unwrap();

    if form.new_password.expose_secret() != form.new_password_check.expose_secret() {
        // [...]
    }
    let username = get_username(user_id, &pool).await.map_err(e500)?;
    // [...]
    todo!()
}
```

```rust
//! src/routes/admin/dashboard.rs
// [...]

#[tracing::instrument(/* */)]
// Marked as `pub`!
pub async fn get_username(/* */) -> Result</* */> {
    // [...]
}
```

We can now pass the username and password combination to `validate_credentials` - if the validation fails, we need to take different actions depending on the returned error:

```rust
//! src/routes/admin/password/post.rs
// [...]
use crate::authentication::{validate_credentials, AuthError, Credentials};

pub async fn change_password(/* */) -> Result</* */> {
    // [...]
    let credentials = Credentials {
        username,
        password: form.0.current_password,
    };
    if let Err(e) = validate_credentials(credentials, &pool).await {
        return match e {
            AuthError::InvalidCredentials(_) => {
                FlashMessage::error("The current password is incorrect.").send();
                Ok(see_other("/admin/password"))
            }
            AuthError::UnexpectedError(_) => Err(e500(e).into()),
        }
    }
    todo!()
}
```

The test should pass.

10.8.2.4 Unhappy Path: The New Password Is Too Short

We do not want our users to choose a weak password - it exposes their account to attackers. OWASP's provides a minimum set of requirements when it comes to password strength - passwords should be longer than 12 characters but shorter than 128 characters.

Add these validation checks to our `POST /admin/password` endpoint as an exercise!

10.8.2.5 Logout

It is finally time to look at the happy path - a user successfully changing their password.
We will use the following scenario to check that everything behaves as expected:

- Log in;
- Change password by submitting the change password form;
- Log out;
- Log in again using the new password.

There is just one roadblock left - we do not have a log-out endpoint yet!
Let's work to bridge this functionality gap before moving forward.

Let's start by encoding our requirements in a test:

```rust
//! tests/api/admin_dashboard.rs
// [...]

#[tokio::test]
async fn logout_clears_session_state() {
    // Arrange
    let app = spawn_app().await;

    // Act - Part 1 - Login
    let login_body = serde_json::json!({
        "username": &app.test_user.username,
        "password": &app.test_user.password
    });
    let response = app.post_login(&login_body).await;
    assert_is_redirect_to(&response, "/admin/dashboard");

    // Act - Part 2 - Follow the redirect
    let html_page = app.get_admin_dashboard_html().await;
    assert!(html_page.contains(&format!("Welcome {}", app.test_user.username)));

    // Act - Part 3 - Logout
    let response = app.post_logout().await;
    assert_is_redirect_to(&response, "/login");

    // Act - Part 4 - Follow the redirect
    let html_page = app.get_login_html().await;
    assert!(html_page.contains(r#"<p><i>You have successfully logged out.</i></p>"#));

    // Act - Part 5 - Attempt to load admin panel
    let response = app.get_admin_dashboard().await;
    assert_is_redirect_to(&response, "/login");
}
```

```rust
//! tests/api/helpers.rs
// [...]

impl TestApp {
    // [...]

    pub async fn post_logout(&self) -> reqwest::Response {
        self.api_client
            .post(&format!("{}/admin/logout", &self.address))
            .send()
            .await
            .expect("Failed to execute request.")
    }
}
```

A log-out is a state-alerting operation: we need to use the `POST` method via a HTML button:

```rust
//! src/routes/admin/dashboard.rs
// [...]

pub async fn admin_dashboard(/* */) -> Result</* */> {
    // [...]
    Ok(HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(
            r#"<!-- [...] -->
<p>Available actions:</p>
<ol>
    <li><a href="/admin/password">Change password</a></li>
    <li>
        <form name="logoutForm" action="/admin/logout" method="post">
            <input type="submit" value="Logout">
        </form>
    </li>
</ol>
<!-- [...] -->"#,
        )))
}
```

We now need to add the corresponding `POST /admin/logout` request handler.

What does it *actually* mean to log out?
We are using session-based authentication - a user is “logged in” if there is a valid user id associated with the `user_id` key in the session state. To log out it is enough to delete the session - remove the state from the storage backend and unset the client-side cookie.
`actix-session` has a dedicated method for this purpose - `Session::purge`. We need to expose it in our `TypedSession` abstraction and then call it in `POST /logout`'s request handler:

```rust
//! src/session_state.rs
// [...]
impl TypedSession {
    // [...]
    pub fn log_out(self) {
        self.0.purge()
    }
}
```

```rust
//! src/routes/admin/logout.rs
use crate::session_state::TypedSession;
use crate::utils::{e500, see_other};
use actix_web::HttpResponse;
use actix_web_flash_messages::FlashMessage;

pub async fn log_out(session: TypedSession) -> Result<HttpResponse, actix_web::Error> {
    if session.get_user_id().map_err(e500)?.is_none() {
        Ok(see_other("/login"))
    } else {
        session.log_out();
        FlashMessage::info("You have successfully logged out.").send();
        Ok(see_other("/login"))
    }
}
```

```rust
//! src/routes/login/get.rs
// [...]
pub async fn login_form(/* */) -> HttpResponse {
    // [...]
    // Display all messages levels, not just errors!
    for m in flash_messages.iter() {
        // [...]
    }
    // [...]
}
```

```rust
//! src/routes/admin/mod.rs
// [...]
mod logout;
pub use logout::log_out;
```

```rust
//! src/startup.rs
use crate::routes::log_out;
// [...]

async fn run(/* */) -> Result<Server, anyhow::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            // [...]
            .route("/admin/logout", web::post().to(log_out))
        // [...]
    })
    // [...]
}
```

10.8.2.6 Happy Path: The Password Was Changed Successfully

We can now get back to the happy path scenario in our change password flow:

- Log in;
- Change password by submitting the change password form;
- Log out;
- Log in again, successfully, using the new password.

Let's add an integration test:

```rust
//! tests/api/change_password.rs
// [...]

#[tokio::test]
async fn changing_password_works() {
    // Arrange
    let app = spawn_app().await;
    let new_password = Uuid::new_v4().to_string();

    // Act - Part 1 - Login
    let login_body = serde_json::json!({
        "username": &app.test_user.username,
        "password": &app.test_user.password
    });
    let response = app.post_login(&login_body).await;
    assert_is_redirect_to(&response, "/admin/dashboard");

    // Act - Part 2 - Change password
    let response = app
        .post_change_password(&serde_json::json!({
            "current_password": &app.test_user.password,
            "new_password": &new_password,
            "new_password_check": &new_password,
        }))
        .await;
    assert_is_redirect_to(&response, "/admin/password");

    // Act - Part 3 - Follow the redirect
    let html_page = app.get_change_password_html().await;
    assert!(html_page.contains("<p><i>Your password has been changed.</i></p>"));

    // Act - Part 4 - Logout
    let response = app.post_logout().await;
    assert_is_redirect_to(&response, "/login");

    // Act - Part 5 - Follow the redirect
    let html_page = app.get_login_html().await;
    assert!(html_page.contains("<p><i>You have successfully logged out.</i></p>"));

    // Act - Part 6 - Login using the new password
    let login_body = serde_json::json!({
        "username": &app.test_user.username,
        "password": &new_password
    });
    let response = app.post_login(&login_body).await;
    assert_is_redirect_to(&response, "/admin/dashboard");
}
```

This is the most complex user scenario we have written so far - a grand total of six steps. This is far from being a record - enterprise applications often require tens of steps to execute real world business processes. It takes a lot of work to keep the test suite readable and maintainable in those scenarios.

The test currently fails at the third step - `POST /admin/password` panics because we left a `todo!()` invocation after the preliminary input validation steps. To implement the required functionality we will need to compute the hash of the new password and then store it in the database - we can add a new dedicated routine to our `authentication` module:

```rust
//! src/authentication.rs
use argon2::password_hash::SaltString;
use argon2::{
    Algorithm, Argon2, Params, PasswordHash,
    PasswordHasher, PasswordVerifier, Version
};
// [...]

#[tracing::instrument(name = "Change password", skip(password, pool))]
pub async fn change_password(
    user_id: uuid::Uuid,
    password: Secret<String>,
    pool: &PgPool,
) -> Result<(), anyhow::Error> {
    let password_hash = spawn_blocking_with_tracing(
            move || compute_password_hash(password)
        )
        .await?
        .context("Failed to hash password")?;
    sqlx::query!(
        r#"
        UPDATE users
        SET password_hash = $1
        WHERE user_id = $2
        "#,
        password_hash.expose_secret(),
        user_id
    )
    .execute(pool)
    .await
    .context("Failed to change user's password in the database.")?;
    Ok(())
}

fn compute_password_hash(
    password: Secret<String>
) -> Result<Secret<String>, anyhow::Error> {
    let salt = SaltString::generate(&mut rand::thread_rng());
    let password_hash = Argon2::new(
        Algorithm::Argon2id,
        Version::V0x13,
        Params::new(15000, 2, 1, None).unwrap(),
    )
    .hash_password(password.expose_secret().as_bytes(), &salt)?
    .to_string();
    Ok(Secret::new(password_hash))
}
```

For `Argon2` we used the parameters recommended by OWASP, the same ones we were already using in our test suite.

We can now plug this function into the request handler:

```rust
//! src/routes/admin/password/post.rs
// [...]
pub async fn change_password(/* */) -> Result</* */> {
    // [...]
    crate::authentication::change_password(user_id, form.0.new_password, &pool)
        .await
        .map_err(e500)?;
    FlashMessage::error("Your password has been changed.").send();
    Ok(see_other("/admin/password"))
}
```

The test should now pass.

10.9 Refactoring

We have added many new endpoints that are restricted to authenticated users. For the sake of speed, we have copy-pasted the same authentication logic across multiple request handlers - it is a good idea to take a step back and try to figure out if we can come up with a better solution.

Let's look at `POST /admin/passwords` as an example. We currently have:

```rust
//! src/routes/admin/password/post.rs
// [...]

pub async fn change_password(/* */) -> Result<HttpResponse, actix_web::Error> {
    let user_id = session.get_user_id().map_err(e500)?;
    if user_id.is_none() {
        return Ok(see_other("/login"));
    };
    let user_id = user_id.unwrap();
    // [...]
}
```

We can factor it out as a new `reject_anonymous_users` function:

```rust
//! src/routes/admin/password/post.rs
use actix_web::error::InternalError;
use uuid::Uuid;
// [...]

async fn reject_anonymous_users(
    session: TypedSession
) -> Result<Uuid, actix_web::Error> {
    match session.get_user_id().map_err(e500)? {
        Some(user_id) => Ok(user_id),
        None => {
            let response = see_other("/login");
            let e = anyhow::anyhow!("The user has not logged in");
            Err(InternalError::from_response(e, response).into())
        }
    }
}

pub async fn change_password(/* */) -> Result<HttpResponse, actix_web::Error> {
    let user_id = reject_anonymous_users(session).await?;
    // [...]
}
```

Notice how we moved the redirect response on the error path in order to use the `?` operator in our request handler.
We could now go and refactor all other `/admin/*` routes to leverage `reject_anonymous_users`.
Or, if you are feeling adventurous, we could try writing a middleware to handle this for us - let's do it!

10.9.1 How To Write An `actix-web` Middleware

Writing a full-blown middleware in `actix-web` can be challenging - it requires us to understand their `Transform` and `Service` traits.
Those abstractions are powerful, but power comes at the cost of complexity.

Our needs are quite simple, we can get away with less: `actix_web_lab::from_fn`.
`actix_web_lab` is a crate used to experiment with future additions to the `actix_web` framework, with a faster release policy. Let's add it to our dependencies:

```
#! Cargo.toml
# [...]
[dependencies]
actix-web-lab = "0.15"
# [...]
```

`from_fn` takes an asynchronous function as argument and returns an `actix-web` middleware as output. The asynchronous function must have the following signature and structure:

```rust
use actix_web_lab::middleware::Next;
use actix_web::body::MessageBody;
use actix_web::dev::{ServiceRequest, ServiceResponse};

async fn my_middleware(
    req: ServiceRequest,
    next: Next<impl MessageBody>,
) -> Result<ServiceResponse<impl MessageBody>, Error> {
    // before the handler is invoked

    // Invoke handler
    let response = next.call(req).await;

    // after the handler was invoked
}
```

Let's adapt `reject_anonymous_users` to follow those requirements - it will live in our authentication module.

```rust
//! src/authentication/mod.rs
mod middleware;
mod password;
pub use password::{
    change_password, validate_credentials,
    AuthError, Credentials
};
pub use middleware::reject_anonymous_users;
```

```rust
//! src/authentication/password.rs
// Copy over **everything** from the old src/authentication.rs
```

This will be our empty canvas:

```rust
//! src/authentication/middleware.rs
use actix_web_lab::middleware::Next;
use actix_web::body::MessageBody;
use actix_web::dev::{ServiceRequest, ServiceResponse};

```

```rust
pub async fn reject_anonymous_users(
    mut req: ServiceRequest,
    next: Next<impl MessageBody>,
) -> Result<ServiceResponse<impl MessageBody>, actix_web::Error> {
    todo!()
}
```

To start out, we need to get our hands on a `TypedSession` instance. `ServiceRequest` is nothing more than a wrapper around `HttpRequest` and `Payload`, therefore we can leverage our existing implementation of `FromRequest`:

```rust
//! src/authentication/middleware.rs
use actix_web_lab::middleware::Next;
use actix_web::body::MessageBody;
use actix_web::dev::{ServiceRequest, ServiceResponse};
use actix_web::FromRequest;
use crate::session_state::TypedSession;

pub async fn reject_anonymous_users(
    mut req: ServiceRequest,
    next: Next<impl MessageBody>,
) -> Result<ServiceResponse<impl MessageBody>, actix_web::Error> {
    let session = {
        let (http_request, payload) = req.parts_mut();
        TypedSession::from_request(http_request, payload).await
    }?;
    todo!()
}
```

Now that we have the session handler, we can check if the session state contains a user id:

```rust
//! src/authentication/middleware.rs
use actix_web::error::InternalError;
use crate::utils::{e500, see_other};
// [...]

pub async fn reject_anonymous_users(
    mut req: ServiceRequest,
    next: Next<impl MessageBody>,
) -> Result<ServiceResponse<impl MessageBody>, actix_web::Error> {
    let session = {
        let (http_request, payload) = req.parts_mut();
        TypedSession::from_request(http_request, payload).await
    }?;

    match session.get_user_id().map_err(e500)? {
        Some(_) => next.call(req).await,
        None => {
            let response = see_other("/login");
            let e = anyhow::anyhow!("The user has not logged in");
            Err(InternalError::from_response(e, response).into())
        }
    }
}
```

This, as it stands, is already useful - it can be leveraged to protect endpoints that require authentication.

At the same time, it isn't equivalent to what we had before - how are we going to access the retrieved user id in our endpoints?

This is a common issue when working with middlewares that extract information out of incoming requests - it is solved via request extensions.
The middleware inserts the information it wants to pass to downstream request handlers into the type map attached to the incoming request (`request.extensions_mut()`).
Request handlers can then access it using the `ReqData` extractor.

Let's start by performing the insertion.
We will define a new-type wrapper, `UserId`, to prevent conflicts in the type map:

```rust
//! src/authentication/mod.rs
// [...]
pub use middleware::UserId;
```

```rust
//! src/authentication/middleware.rs
use uuid::Uuid;
use std::ops::Deref;
use actix_web::HttpMessage;
// [...]

#[derive(Copy, Clone, Debug)]
pub struct UserId(Uuid);

impl std::fmt::Display for UserId {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        self.0.fmt(f)
    }
}

impl Deref for UserId {
    type Target = Uuid;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

pub async fn reject_anonymous_users(/* */) -> Result</* */> {
    // [...]
    match session.get_user_id().map_err(e500)? {
        Some(user_id) => {
            req.extensions_mut().insert(UserId(user_id));
            next.call(req).await
        }
        None => // [...]
    }
}
```

We can now access it in `change_password`:

```rust
//! src/routes/admin/password/post.rs
use crate::authentication::UserId;
// [...]

pub async fn change_password(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
    // No longer injecting TypedSession!
    user_id: web::ReqData<UserId>,
) -> Result<HttpResponse, actix_web::Error> {
    let user_id = user_id.into_inner();
    // [...]
    let username = get_username(*user_id, &pool).await.map_err(e500)?;
    // [...]
    crate::authentication::change_password(*user_id, form.0.new_password, &pool)
        .await
        .map_err(e500)?;
    // [...]
}
```

If you run the test suite, you'll be greeted by several failures. If you inspect the logs for one of them, you'll find the following error:

```
Error encountered while processing the incoming HTTP request:
"Missing expected request extension data"
```

It makes sense - we never registered our middleware against our `App` instance, therefore the insertion of `UserId` into the request extensions never takes place.
Let's fix it.
Our routing table currently looks like this:

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

async fn run(/* */) -> Result<Server, anyhow::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            .wrap(message_framework.clone())
            .wrap(SessionMiddleware::new(
                redis_store.clone(),
                secret_key.clone(),
            ))
            .wrap(TracingLogger::default())
            .route("/", web::get().to(home))
            .route("/login", web::get().to(login_form))
            .route("/login", web::post().to(login))
            .route("/health_check", web::get().to(health_check))
            .route("/newsletters", web::post().to(publish_newsletter))
            .route("/subscriptions", web::post().to(subscribe))
            .route("/subscriptions/confirm", web::get().to(confirm))
            .route("/admin/dashboard", web::get().to(admin_dashboard))
            .route("/admin/password", web::get().to(change_password_form))
            .route("/admin/password", web::post().to(change_password))
            .route("/admin/logout", web::post().to(log_out))
            .app_data(db_pool.clone())
            .app_data(email_client.clone())
            .app_data(base_url.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

We want to apply our middleware logic exclusively to `/admin/*` endpoints, but calling `wrap` on `App` would apply the middleware to all our routes.
Considering that our target endpoints all share the same common base path, we can achieve our objective by introducing a **scope**:

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

async fn run(/* */) -> Result<Server, anyhow::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
            .wrap(message_framework.clone())
            .wrap(SessionMiddleware::new(
                redis_store.clone(),
                secret_key.clone(),
            ))
            .wrap(TracingLogger::default())
            .route("/", web::get().to(home))
            .route("/login", web::get().to(login_form))
            .route("/login", web::post().to(login))
            .route("/health_check", web::get().to(health_check))
            .route("/newsletters", web::post().to(publish_newsletter))
            .route("/subscriptions", web::post().to(subscribe))
            .route("/subscriptions/confirm", web::get().to(confirm))
            .service(
                web::scope("/admin")
                    .route("/dashboard", web::get().to(admin_dashboard))
                    .route("/password", web::get().to(change_password_form))
                    .route("/password", web::post().to(change_password))
                    .route("/logout", web::post().to(log_out)),
            )
            .app_data(db_pool.clone())
            .app_data(email_client.clone())
            .app_data(base_url.clone())
    })
    .listen(listener)?
    .run();
    Ok(server)
}
```

We can now add a middleware restricted to `/admin/*` by calling `wrap` on `web::scope("admin")` instead of the top-level `App`:

```rust
//! src/startup.rs
use crate::authentication::reject_anonymous_users;
use actix_web_lab::middleware::from_fn;
// [...]

async fn run(/* */) -> Result<Server, anyhow::Error> {
    // [...]
    let server = HttpServer::new(move || {
        App::new()
```

```
            .wrap(message_framework.clone())
            .wrap(SessionMiddleware::new(
                redis_store.clone(),
                secret_key.clone(),
            ))
            .wrap(TracingLogger::default())
            // [...]
            .service(
                web::scope("/admin")
                    .wrap(from_fn(reject_anonymous_users))
                    .route("/dashboard", web::get().to(admin_dashboard))
                    .route("/password", web::get().to(change_password_form))
                    .route("/password", web::post().to(change_password))
                    .route("/logout", web::post().to(log_out)),
            )
            // [...]
    })
    // [...]
}
```

If you run the test suite, it should pass (apart from our idempotency test).
You can now go through the other `/admin/*` endpoints and remove the duplicated check-if-logged-in-or-redirect code.

10.10 Summary

Take a deep breath - we covered **a lot** of ground in this chapter.
We built, from scratch, a large chunk of the machinery that powers authentication in most of the software you interact with on a daily basis.

API security is an amazingly broad topic - we explored together a selection of key techniques, but this introduction is in no way exhaustive. There are entire areas that we just mentioned but did not have a chance to cover in depth (e.g. OAuth2/OpenID Connect). Look at the bright side - you learned enough to go and tackle those topics on your own should your applications require them.

It is easy to forget the bigger picture when you spend a lot of time working close to the details - why did we even start to talk about API security?
That's right! We had just built a new endpoint to send out newsletter issues and we did not want to give everyone on the Internet a chance to broadcast content to our audience. We added 'Basic' authentication to `POST /newsletters` early in the chapter but we have not yet ported it over to session-based authentication.

As an exercise, *before engaging with the new chapter*, do the following:

- Add a `Send a newsletter issue` link to the admin dashboard;
- Add an HTML form at `GET /admin/newsletters` to submit a new issue;
- Adapt `POST /newsletters` to process the form data:
  – Change the route to `POST /admin/newsletters`;
  – Migrate from 'Basic' to session-based authentication;
  – Use the `Form` extractor (`application/x-www-form-urlencoded`) instead of the `Json` extractor (`application/json`) to handle the request body;

  - Adapt the test suite.

It will take a bit of work but - and that's the key here - you *know* how to do all these things. We have done them together before - feel free to go back to the relevant sections as you progress through the exercise.

> On GitHub you can find a project snapshot before and after fulfilling the exercise requirements. The next chapter assumes that the exercise has been completed - make sure to double-check your solution before moving forward!

`POST /admin/newsletters` will be under the spotlight during the next chapter - we will be reviewing our initial implementation under a microscope to understand how it behaves when things break down. It will give us a chance to talk more broadly about fault tolerance, scalability and asynchronous processing.

Chapter 11

Fault-tolerant Workflows

We kept the first iteration of our newsletter endpoint very simple: emails are immediately sent out to all subscribers via Postmark, one API call at a time.
This is good enough if the audience is small - it breaks down, in a variety of ways, when dealing with hundreds of subscribers.

We want our application to be **fault-tolerant**.
Newsletter delivery should not be disrupted by transient failures like application crashes, Postmark API errors or network timeouts. To deliver a reliable service in the face of failure we will have to explore new concepts: idempotency, locking, queues and background jobs.

11.1 `POST /admin/newsletters` - A Refresher

Let's refresh our memory before jumping straight into the task: what does `POST /admin/newsletters` look like?[1]

The endpoint is invoked when a logged-in newsletter author submits the HTML form served at `GET /admin/newsletters`.
We parse the form data out of the HTTP request body and, if nothing is amiss, kick-off the processing.

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

#[derive(serde::Deserialize)]
pub struct FormData {
    title: String,
    text_content: String,
    html_content: String,
}

#[tracing::instrument(/* */)]
pub async fn publish_newsletter(
```

[1] At the end of chapter 10 you were asked to convert `POST /newsletters` (JSON + 'Basic' auth) into `POST /admin/newsletters` (HTML Form data + session-based auth) as a take-home exercise. Your implementation might differ slightly from mine, therefore the code blocks here might not match exactly what you see in your IDE. Check the book's GitHub repository to compare solutions.

```rust
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
    email_client: web::Data<EmailClient>,
) -> Result<HttpResponse, actix_web::Error> {
    // [...]
}
```

We start by fetching all confirmed subscribers from our Postgres database.

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

#[tracing::instrument(/* */)]
pub async fn publish_newsletter(/* */) -> Result<HttpResponse, actix_web::Error> {
    // [...]
    let subscribers = get_confirmed_subscribers(&pool).await.map_err(e500)?;
    // [...]
}

struct ConfirmedSubscriber {
    email: SubscriberEmail,
}

#[tracing::instrument(/* */)]
async fn get_confirmed_subscribers(
    pool: &PgPool,
) -> Result<Vec<Result<ConfirmedSubscriber, anyhow::Error>>, anyhow::Error> {
    /* */
}
```

We iterate over the retrieved subscribers, sequentially.
For each user, we try to send out an email with the new newsletter issue.

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

#[tracing::instrument(/* */)]
pub async fn publish_newsletter(/* */) -> Result<HttpResponse, actix_web::Error> {
    // [...]
    let subscribers = get_confirmed_subscribers(&pool).await.map_err(e500)?;
    for subscriber in subscribers {
        match subscriber {
            Ok(subscriber) => {
                email_client
                    .send_email(/* */)
                    .await
                    .with_context(/* */)
                    .map_err(e500)?;
            }
            Err(error) => {
                tracing::warn!(/* */);
            }
        }
    }
    FlashMessage::info("The newsletter issue has been published!").send();
    Ok(see_other("/admin/newsletters"))
}
```

Once all subscribers have been taken care of, we redirect the author back to the newsletter form - they will be shown a flash message confirming that the issue was published successfully.

11.2 Our Goal

We want to ensure **best-effort delivery**: we strive to deliver the new newsletter issue to all subscribers.
We cannot *guarantee* that all emails will be delivered: some accounts might just have been deleted.

At the same time, we should try to minimize duplicates - i.e. a subscriber receiving the same issue multiple times. We cannot rule out duplicates entirely (we will later discuss why), but our implementation should minimize their frequency.

11.3 Failure Modes

Let's have a look at the possible failure modes of our `POST /admin/newsletters` endpoint.
Can we still achieve best-effort delivery when something goes awry?

11.3.1 Invalid Inputs

There might be issues with the incoming request: the body is malformed or the user has not authenticated.
Both scenarios are already handled appropriately:

- the `web::Form` extractor returns a `400 Bad Request`[2] if the incoming form data is invalid;
- unauthenticated users are redirected back to the login form.

11.3.2 Network I/O

Problems might arise when we interact with other machines over the network.

11.3.2.1 Postgres

The database might misbehave when we try to retrieve the current list of subscribers. We do not have a lot of options apart from retrying. We can:

- retry in process, by adding some logic around the `get_confirmed_subscribers` call;
- give up by returning an error to the user. The user can then decide if they want to retry or not.

The first option makes our application more resilient to spurious failures. Nonetheless, you can only perform a finite number of retries; you will have to give up eventually.

[2] It is up for debate if this is actually the best way to handle an invalid body. Assuming no mistakes were made on our side, submitting the HTML form we serve on `GET /admin/newsletters` should always result into a request body that passes the basic validation done by the `Json` extractor - a.k.a. we get all the fields we expect. But mistakes are a possibility - we cannot rule out that some of the types used in `FormData` as fields might start doing more advanced validation in the future - it'd be safer to redirect the user back to the form page with a proper error message when body validation fails. You can try it out as an exercise.

Our implementation opts for the second strategy from the get-go. It might result in a few more 500s, but it is not incompatible with our over-arching objective.

11.3.2.2 Postmark - API Errors

What about email delivery issues?
Let's start with the simplest scenario: Postmark returns an error when we try to email one of our subscribers.
Our current implementation bails out: we abort the processing and return a `500 Internal Server Error` to the caller.

We are sending emails out sequentially. We will never get a chance to deliver the new issue to the subscribers at the end of the list if we abort as soon as an API error is encountered. This is far from being "best-effort delivery".

This is not the end of our problems either - can the newsletter author retry the form submission?

It depends on **where** the error occurred.

Was it the first subscriber in the list returned by our database query?
No problem, nothing has happened yet.

What if it were the third subscriber in the list? Or the fifth? Or the one-hundredth?
We have a problem: some subscribers have been sent the new issue, others haven't.
If the author retries, some subscribers are going to receive the issue **twice**.
If they don't retry, some subscribers might never receive the issue.

Damned if you do, damned if you don't.

You might recognize the struggle: we are dealing with a **workflow**, a combination of multiple **sub-tasks**.
We faced something similar in chapter 7 when we had to execute a sequence of SQL queries to create a new subscriber. Back then, we opted for an all-or-nothing semantics using SQL transactions: nothing happens unless all queries succeed. Postmark's API does not provide any[3] kind of transactional semantics - each API call is its own unit of work, we have no way to link them together.

11.3.3 Application Crashes

Our application could crash at any point in time. It might, for example, run out of memory or the server it is running on might be abruptly terminated (welcome to the cloud!).

A crash, in particular, might happen **after** we started to process the subscribers list but **before** we got to the end of it. The author will receive an error message in the browser.
Re-submitting the form is likely to result in a high number of redundant deliveries, just like we observed when discussing the consequences of Postmark's API errors.

[3] Postmark provides a batch email API - it is not clear, from their documentation, if they retry messages within a batch to ensure best-effort delivery. Regardless, there is a maximum batch size (500) - if your audience is big enough you have to think about how to batch batches: back to square zero. From a learning perspective, we can safely ignore their batch API entirely.

11.3.4 Author Actions

Last but not least, we might have issues in the interaction between the author and the API.

If we are dealing with a large audience, it might take minutes to process the entire subscribers list. The author might get impatient and choose to re-submit the form. The browser might decide to give up (client-side timeout). Or, equally problematic, the author might click on the `Submit` button more than once[4], by mistake.

Once again, we end up in a corner because our implementation is not retry-safe.

11.4 Idempotency: An Introduction

`POST /admin/newsletters` is, all things considered, a pretty simple endpoint. Nonetheless, our investigation highlighted several scenarios where the current implementation fails to meet our expectations.
Most of our problems boil down to a specific limitation: **it is not safe to retry**.

Retry-safety has a dramatic impact on the ergonomics of an API. It is substantially easier to write a reliable API client if you can safely retry when something goes wrong.
But what does retry-safety **actually** entail?

We built an intuitive understanding of what it means in our domain, newsletter delivery - send the content to every subscriber no more than once. How does that transfer to another domain?
You might be surprised to find out that we do not have a clear industry-accepted definition. It is a tricky subject.
For the purpose of this book, we will define retry-safety as follows:

> An API endpoint is retry-safe (or **idempotent**) if the caller has no way to **observe** if a request has been sent to the server once or multiple times.

We will probe and explore this definition for a few sections: it is important to fully understand its ramifications.

If you have been in the industry long enough, you have probably heard another term used to describe the concept of retry-safety: **idempotency**. They are mostly used as synonyms - we will use idempotency going forward, mostly to align with other industry terminology that will be relevant to our implementation (i.e. idempotency keys).

11.4.1 Idempotency In Action: Payments

Let's explore the implications of our idempotency definition in another domain, payments.
Our fictional payments API exposes three endpoints:

- `GET /balance`, to retrieve your current account balance;
- `GET /payments`, to retrieve the list of payments you initiated;
- `POST /payments`, to initiate a new payment.

[4] Client-side JavaScript can be used to disable buttons after they have been clicked, reducing the likelihood of this scenario.

`POST /payments`, in particular, takes as input the beneficiary details and the payment amount. An API call triggers a money transfer from your account to the specified beneficiary; your balance is reduced accordingly (i.e. `new_balance = old_balance - payment_amount`).

Let's consider this scenario: your balance is 400 USD and you send a request to transfer 20 USD. The request succeeds: the API returned a `200 OK`[5], your balance was updated to 380 USD and the beneficiary received 20 USD.
You then retry the same request - e.g. you click twice on the `Pay now` button.
What should happen if `POST /payments` is idempotent?

Our idempotency definition is built around the concept of *observability* - properties of the system state that the caller can inspect by interacting with the system itself.
For example: you could easily determine that the second call is a retry by going through the logs emitted by the API. But the caller is not an operator - they have no way to inspect those logs. They are *invisible* to the users of the API - in so far as idempotency is concerned, they don't exist. They are not part of the **domain model** exposed and manipulated by the API.

The domain model in our example includes:

- the caller's account, with its balance (via `GET /balance`) and payment history (via `GET /payments`);
- other accounts[6] reachable over the payment network (i.e. beneficiaries we can pay).

Given the above, we can say that `POST /payments` is idempotent if, when the request is retried,

- the balance remains 380 USD;
- no additional money are transferred to the beneficiary.

There is one more detail to sort out - what HTTP response should the server return for the retried request?
The caller should not be able to observe that the second request was a retry. The payment succeeded, therefore the server should return a success response that is semantically equivalent to the HTTP response used to answer the initial request.

11.4.2 Idempotency Keys

There is room for ambiguity in our definition of idempotency: how do we distinguish between a retry and a user trying to perform two distinct payments for the same amount to the same beneficiary?

We need to understand the caller's **intent**.

We could try to use a heuristic - e.g. the second request is a duplicate if it was sent no more than 5 minutes later.
This could be a good starting point, but it is not bulletproof. The consequences of misclassification could be dire, both for the caller and our reputation as an organization (e.g. a late retry causing

[5] Real-world payment systems would most likely return a `202 Accepted` - payment authorization, execution and settlement happen at different points in time. We are keeping things simple for the sake of our example.

[6] The other accounts on the payment network are not exposed to us via the API (i.e. we can't query for John's payment history), but we can still observe if one of our payments did or did not reach its beneficiary - e.g. by calling them or if they reach out to us to complain they haven't received the money yet! API calls can have a material effect on the physical world around us - that's why this whole computer thing is so powerful and scary at the same time!

a double payment).

Given that this is all about understanding the caller's intent, there is no better strategy than empowering the caller themselves to tell us what they are trying to do. This is commonly accomplished using **idempotency keys**.
The caller generates a unique identifier, the idempotency key, for every state-altering operation they want to perform. The idempotency key is attached to the outgoing request, usually as an HTTP header (e.g. `Idempotency-Key`[7]).
The server can now easily spot duplicates:

- two identical requests, different idempotency keys = two distinct operations;
- two identical requests, same idempotency key = a single operation, the second request is a duplicate;
- two different[8] requests, same idempotency key = the first request is processed, the second one is rejected.

We will start requiring an idempotency key in `POST /admin/newsletters` as part of our idempotency implementation.

11.4.3 Concurrent Requests

What should happen when two duplicate requests are fired **concurrently** - i.e. the second request reaches the server before it finishes processing the first one?

We do not yet know the outcome of the first request. Processing both requests in parallel might also introduce the risk of performing side effects more than once (e.g. initiating two distinct payments).

It is common to introduce **synchronization**: the second request should not be processed until the first one has completed.
We have two options:

- Reject the second request by returning a `409 Conflict` status code back to the caller;
- Wait until the first request completes processing. Then return the same response back to the caller.

Both are viable.
The latter is fully transparent to the caller, making it easier to consume the API - they don't have to handle yet another transient failure mode. There is a price[9] to pay though: both the client and the server need to keep an open connection while spinning idle, waiting for the other task to complete.
Considering our use case (processing forms), we will go for the second strategy in order to minimize the number of user-visible errors - browsers do not automatically retry `409`s.

[7] An `Internet-Draft` in the `httpapi` IETF working group proposed to standardize the `Idempotency-Key` header, but the conversation does not seem to have moved forward (the draft expired in January 2022).

[8] We are not actually going to implement a likeness check for incoming requests - it can get quite tricky: do the headers matter? All of them? A subset of them? Does the body need to match byte-by-byte? Is it enough if it's semantically equivalent (e.g. two JSON objects with identical keys and values)? It can be done, but it is beyond the scope of this chapter.

[9] If we are being pessimistic, this could be abused to mount a denial of service attack against the API. It can be avoided by enforcing fair-usage limitations - e.g. it's OK to have a handful of concurrent requests for the same idempotency key, but the server will start returning errors if we end up dealing with tens of duplicates.

11.5 Requirements As Tests #1

Let's start by focusing on the simplest scenario: a request was received and processed successfully, then a retry is performed.
We expect a success response with no duplicate newsletter delivery:

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn newsletter_creation_is_idempotent() {
    // Arrange
    let app = spawn_app().await;
    create_confirmed_subscriber(&app).await;
    app.test_user.login(&app).await;

    Mock::given(path("/email"))
        .and(method("POST"))
        .respond_with(ResponseTemplate::new(200))
        .expect(1)
        .mount(&app.email_server)
        .await;

    // Act - Part 1 - Submit newsletter form
    let newsletter_request_body = serde_json::json!({
        "title": "Newsletter title",
        "text_content": "Newsletter body as plain text",
        "html_content": "<p>Newsletter body as HTML</p>",
        // We expect the idempotency key as part of the
        // form data, not as an header
        "idempotency_key": uuid::Uuid::new_v4().to_string()
    });
    let response = app.post_publish_newsletter(&newsletter_request_body).await;
    assert_is_redirect_to(&response, "/admin/newsletters");

    // Act - Part 2 - Follow the redirect
    let html_page = app.get_publish_newsletter_html().await;
    assert!(
        html_page.contains("<p><i>The newsletter issue has been published!</i></p>")
    );

    // Act - Part 3 - Submit newsletter form **again**
    let response = app.post_publish_newsletter(&newsletter_request_body).await;
    assert_is_redirect_to(&response, "/admin/newsletters");

    // Act - Part 4 - Follow the redirect
    let html_page = app.get_publish_newsletter_html().await;
    assert!(
        html_page.contains("<p><i>The newsletter issue has been published!</i></p>")
    );

    // Mock verifies on Drop that we have sent the newsletter email **once**
}
```

`cargo test` should fail:

```
thread 'newsletter::newsletter_creation_is_idempotent' panicked at
'Verifications failed:
- Mock #1.
    Expected range of matching incoming requests: == 1
    Number of matched incoming requests: 2
[...]'
```

The retry succeeded, but it resulted in the newsletter being delivered twice to our subscriber - the problematic behaviour we identified during the failure analysis at the very beginning of this chapter.

11.6 Implementation Strategies

How do we prevent the retried request from dispatching a new round of emails to our subscribers? We have two options - one requires state, the other doesn't.

11.6.1 Stateful Idempotency: Save And Replay

In the stateful approach, we process the first request and then store its idempotency key next to the HTTP response we are about to return. When a retry comes in, we look for a match in the store against its idempotency key, fetch the saved HTTP response and return it to the caller.
The entire handler logic is short-circuited - it never gets executed. Postmark's API is never called again, preventing duplicate deliveries.

11.6.2 Stateless Idempotency: Deterministic Key Generation

The stateless approach tries to achieve the same outcome without relying on persistence.
For every subscriber, we **deterministically** generate a new idempotency key using their subscriber id, the newsletter content[10] and the idempotency key attached to the incoming request. Every time we call Postmark to send an email we make sure to pass along the subscriber-specific idempotency key.
When a retry comes in, we execute the same processing logic - this leads to the same sequence of HTTP calls to Postmark, using exactly the same idempotency keys. Assuming their idempotency implementation is sound, no new email is going to be dispatched.

11.6.3 Time Is a Tricky Beast

The stateless and the stateful approach are not 100% equivalent.

Let's consider what happens, for example, when a new person subscribes to our newsletter between the initial request and the following retry.

[10] Two different newsletter issues should not generate the same subscriber-specific idempotency key. If that were to happen, you wouldn't be able to send two different issues one after the other because Postmark's idempotency logic would prevent the second set of emails from going out. This is why we must include a fingerprint of the incoming request content in the generation logic for the subscriber-specific idempotency key - it ensures a unique outcome for each subscriber-newsletter issue pair. Alternatively, we must implement a likeness check to ensure that the same idempotency key cannot be used for two different requests to `POST /admin/newsletters` - i.e. the idempotency key is enough to ensure that the newsletter content is not the same.

The stateless approach executes the handler logic in order to process the retried request. In particular, it re-generates the list of *current* subscribers before kicking off the email dispatching `for` loop. As a result, the new subscriber will receive the newsletter issue.
This is not the case when following the stateful approach - we retrieve the HTTP response from the store and return it to the caller without performing any kind of processing.

This is a symptom of a deeper discrepancy - the **elapsed time** between the initial request and the following retry affects the processing outcome when following the stateless approach.
We cannot execute our handler logic against the same snapshot of the state seen by the first request - therefore, the view of the world in the stateless approach is impacted by all the operations that have been committed since the first request was processed[11] (e.g. new subscribers joining the mailing list).

Whether this is acceptable or not depends on the domain.
In our case, the fallout is quite minor - we are just sending extra newsletters out. We could live with it if the stateless approach led us to a dramatically simpler implementation.

11.6.4 Making A Choice

Unfortunately, the circumstances leave us with no wiggle room: Postmark's API does not provide any idempotency mechanism therefore we cannot follow the stateless approach.
The stateful approach happens to be trickier to implement - rejoice, we'll have a chance to learn some new patterns!

11.7 Idempotency Store

11.7.1 Which Database Should We Use?

For each idempotency key, we must store the associated HTTP response.
Our application currently uses two different data sources:

- Redis, to store the session state for each user;
- Postgres, for everything else.

We do not want to store idempotency keys forever - it would be impractical and wasteful.
We also do not want actions performed by a user A to influence the outcome of actions performed by user B - there is a concrete security risk (cross-user data leakage) if proper isolation is not enforced.

Storing idempotency keys and responses into the session state of the user would guarantee both isolation and expiry out of the box. At the same time, it doesn't feel right to tie the lifespan of idempotency keys to the lifespan of the corresponding user sessions.

Based on our current requirements, Redis looks like the best solution to store our `(user_id, idempotency_key, http_response)` triplets. They would have their own time-to-live policy, with no ties to session states, and Redis would take care of cleaning old entries for us.

Unfortunately, new requirements will soon emerge and turn Redis into a limiting choice. There is not much to learn by taking the wrong turn here, so I'll cheat and force our hand towards Postgres.

[11] This is equivalent to a non-repeatable read in a relational database.

Spoiler: we will leverage the possibility of modifying the idempotency triplets and our application state within a single SQL transaction.

11.7.2 Schema

We need to define a new table to store the following information:

- user id;
- idempotency key;
- HTTP response.

The user id and the idempotency key can be used as a composite primary key. We should also record when each row was created in order to evict old idempotency keys.

There is a major unknown though: what type should be used to store HTTP responses?

We could treat the whole HTTP response as a blob of bytes, using `bytea` as column type.
Unfortunately, it'd be tricky to re-hydrate the bytes into an `HttpResponse` object - `actix-web` does not provide any serialization/deserialization implementation for `HttpResponse`.
We are going to write our own (de)serialisation code - we will work with the core components of an HTTP response:

- status code;
- headers;
- body.

We are not going to store the HTTP version - the assumption is that we are working exclusively with `HTTP/1.1`.
We can use `smallint` for the status code - it's maximum value is 32767, which is more than enough. `bytea` will do for the body.
What about headers? What is their type?

We can have multiple header values associated to the same header name, therefore it makes sense to represent them as an array of `(name, value)` pairs.
We can use `TEXT` for the `name` (see `http`'s implementation) while `value` will require `BYTEA` because it allows opaque octets (see `http`'s test cases).
Postgres does not support arrays of tuples, but there is a workaround: we can define a Postgres **composite type** - i.e. a named collection of fields, the equivalent of a struct in our Rust code.

```sql
CREATE TYPE header_pair AS (
    name TEXT,
    value BYTEA
);
```

We can now put together the migration script:

```bash
sqlx migrate add create_idempotency_table
```

```sql
-- migrations/20220211080603_create_idempotency_table.sql
CREATE TYPE header_pair AS (
    name TEXT,
    value BYTEA
);
```

```sql
CREATE TABLE idempotency (
   user_id uuid NOT NULL REFERENCES users(user_id),
   idempotency_key TEXT NOT NULL,
   response_status_code SMALLINT NOT NULL,
   response_headers header_pair[] NOT NULL,
   response_body BYTEA NOT NULL,
   created_at timestamptz NOT NULL,
   PRIMARY KEY(user_id, idempotency_key)
);
```

```
sqlx migrate run
```

We could have defined an overall `http_response` composite type, but we would have run into a bug in `sqlx` which is in turn caused by a bug in the Rust compiler. Best to avoid nested composite types for the time being.

11.8 Save And Replay

11.8.1 Read Idempotency Key

Our `POST /admin/newsletters` endpoint is being triggered by an HTML form submission, therefore we do not have control over the headers that are being sent to the server.
The most practical choice is to embed the idempotency key inside the form data:

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

#[derive(serde::Deserialize)]
pub struct FormData {
    title: String,
    text_content: String,
    html_content: String,
    // New field!
    idempotency_key: String
}
```

We do not care about the exact format of the idempotency key, as long as it's not empty and it's reasonably long.
Let's define a new type to enforce minimal validation:

```rust
//! src/lib.rs
// [...]
// New module!
pub mod idempotency;
```

```rust
//! src/idempotency/mod.rs
mod key;
pub use key::IdempotencyKey;
```

```rust
//! src/idempotency/key.rs
#[derive(Debug)]
pub struct IdempotencyKey(String);

impl TryFrom<String> for IdempotencyKey {
    type Error = anyhow::Error;
    fn try_from(s: String) -> Result<Self, Self::Error> {
        if s.is_empty() {
            anyhow::bail!("The idempotency key cannot be empty");
        }
        let max_length = 50;
        if s.len() >= max_length {
            anyhow::bail!(
                "The idempotency key must be shorter
                than {max_length} characters");
        }
        Ok(Self(s))
    }
}

impl From<IdempotencyKey> for String {
    fn from(k: IdempotencyKey) -> Self {
        k.0
    }
}

impl AsRef<str> for IdempotencyKey {
    fn as_ref(&self) -> &str {
        &self.0
    }
}
```

We can now use it in `publish_newsletter`:

```rust
//! src/utils.rs
use actix_web::http::StatusCode;
// [...]

// Return a 400 with the user-representation of the validation error as body.
// The error root cause is preserved for logging purposes.
pub fn e400<T: std::fmt::Debug + std::fmt::Display>(e: T) -> actix_web::Error
where
    T: std::fmt::Debug + std::fmt::Display + 'static
{
    actix_web::error::ErrorBadRequest(e)
}
```

```rust
//! src/routes/admin/newsletter/post.rs
use crate::idempotency::IdempotencyKey;
use crate::utils::e400;
// [...]

pub async fn publish_newsletter(/* */) -> Result<HttpResponse, actix_web::Error> {
    // We must destructure the form to avoid upsetting the borrow-checker
    let FormData { title, text_content, html_content, idempotency_key } = form.0;
    let idempotency_key: IdempotencyKey = idempotency_key.try_into().map_err(e400)?;
    let subscribers = get_confirmed_subscribers(&pool).await.map_err(e500)?;
    for subscriber in subscribers {
        match subscriber {
            Ok(subscriber) => {
                // No longer using `form.<X>`
                email_client
```

```
                    .send_email(&subscriber.email, &title, &html_content, &text_content)
                    // [...]
            }
            // [...]
        }
    }
    // [...]
}
```

Success! The idempotency key has been parsed and validated.
Some of our old tests, though, are not particularly happy:

```
thread 'newsletter::you_must_be_logged_in_to_publish_a_newsletter'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `303`'

thread 'newsletter::newsletters_are_not_delivered_to_unconfirmed_subscribers'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `303`'

thread 'newsletter::newsletters_are_delivered_to_confirmed_subscribers'
panicked at 'assertion failed: `(left == right)`
  left: `400`,
 right: `303`'
```

Our test requests are being rejected because they do not include an idempotency key.
Let's update them:

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn newsletters_are_not_delivered_to_unconfirmed_subscribers() {
    // [...]
    let newsletter_request_body = serde_json::json!({
        // [...]
        "idempotency_key": uuid::Uuid::new_v4().to_string()
    });
}

#[tokio::test]
async fn newsletters_are_delivered_to_confirmed_subscribers() {
    // [...]
    let newsletter_request_body = serde_json::json!({
        // [...]
        "idempotency_key": uuid::Uuid::new_v4().to_string()
    });
}

#[tokio::test]
async fn you_must_be_logged_in_to_publish_a_newsletter() {
    // [...]
    let newsletter_request_body = serde_json::json!({
        // [...]
        "idempotency_key": uuid::Uuid::new_v4().to_string()
    });
    // [...]
}
```

Those three tests should now pass again, leaving `newsletter::newsletter_creation_is_idempotent` as the only failing test.

We also need to update `GET /admin/newsletters` to embed a randomly-generated idempotency key in the HTML form:

```rust
//! src/routes/admin/newsletter/get.rs
// [...]

pub async fn publish_newsletter_form(/* */) -> Result<HttpResponse, actix_web::Error> {
    // [...]
    let idempotency_key = uuid::Uuid::new_v4();
    Ok(HttpResponse::Ok()
        .content_type(ContentType::html())
        .body(format!(
            r#"<!-- ... -->
    <form action="/admin/newsletters" method="post">
        <!-- ... -->
        <input hidden type="text" name="idempotency_key" value="{idempotency_key}">
        <button type="submit">Publish</button>
    </form>
    <!-- ... -->"#,
        )))
}
```

11.8.2 Retrieve Saved Responses

The next step is trying to fetch a saved HTTP response from the store, assuming one exists.

It boils down to a single SQL query:

```rust
//! src/idempotency/mod.rs
// [...]
mod persistence;
pub use persistence::get_saved_response;
```

```rust
//! src/idempotency/persistence.rs
use super::IdempotencyKey;
use actix_web::HttpResponse;
use sqlx::PgPool;
use uuid::Uuid;

pub async fn get_saved_response(
    pool: &PgPool,
    idempotency_key: &IdempotencyKey,
    user_id: Uuid,
) -> Result<Option<HttpResponse>, anyhow::Error> {
    let saved_response = sqlx::query!(
        r#"
        SELECT
            response_status_code,
            response_headers,
            response_body
        FROM idempotency
        WHERE
          user_id = $1 AND
          idempotency_key = $2
        "#,
        user_id,
        idempotency_key.as_ref()
    )
    .fetch_optional(pool)
    .await?;
    todo!()
}
```

There is a caveat - `sqlx` does not know how to handle our custom `header_pair` type:

```
error: unsupported type _header_pair of column #2 ("response_headers")
 |
 |       let saved_response = sqlx::query!(
 |  __________________________^
 | |         r#"
 | |         SELECT
.. |
 | |         idempotency_key.as_ref()
 | |     )
   | |_____^
```

It might not be supported out of the box, but there is a mechanism for us to specify how it should be handled - the `Type`, `Decode` and `Encode` traits.
Luckily enough, we do not have to implement them manually - we can derive them with a macro!
We just need to specify the type fields and the name of the composite type as it appears in Postgres; the macro should take care of the rest:

```
//! src/idempotency/persistence.rs
// [...]

#[derive(Debug, sqlx::Type)]
#[sqlx(type_name = "header_pair")]
struct HeaderPairRecord {
    name: String,
    value: Vec<u8>,
}
```

Unfortunately, the error is still there.

```
error: unsupported type _header_pair of column #2 ("response_headers")
 |
 |       let saved_response = sqlx::query!(
 |  __________________________^
 | |         r#"
 | |         SELECT
.. |
 | |         idempotency_key.as_ref()
 | |     )
 | |_____^

// [...] <new error> [...]
```

It turns out that `sqlx::query!` does not handle custom type automatically - we need to explain how we want the custom column to be handled by using an explicit type annotation.
The query becomes:

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn get_saved_response(/* */) -> Result</* */> {
    let saved_response = sqlx::query!(
        r#"
        SELECT
            response_status_code,
            response_headers as "response_headers: Vec<HeaderPairRecord>",
            response_body
        // [...]
        "#,
        // [...]
    )
    // [...]
}
```

At last, it compiles!
Let's map the retrieved data back into a proper `HttpResponse`:

```rust
//! src/idempotency/persistence.rs
use actix_web::http::StatusCode;
// [...]

pub async fn get_saved_response(/* */) -> Result<Option<HttpResponse>, anyhow::Error> {
    let saved_response = sqlx::query!(/* */)
        .fetch_optional(pool)
        .await?;
    if let Some(r) = saved_response {
        let status_code = StatusCode::from_u16(
            r.response_status_code.try_into()?
        )?;
        let mut response = HttpResponse::build(status_code);
        for HeaderPairRecord { name, value } in r.response_headers {
            response.append_header((name, value));
        }
        Ok(Some(response.body(r.response_body)))
    } else {
        Ok(None)
    }
}
```

We can now plug `get_saved_response` into our request handler:

```rust
//! src/routes/admin/newsletter/post.rs
// [...]
use crate::idempotency::get_saved_response;

pub async fn publish_newsletter(
    // [...]
    // Inject the user id extracted from the user session
    user_id: ReqData<UserId>,
) -> Result<HttpResponse, actix_web::Error> {
    let user_id = user_id.into_inner();
    let FormData {
        title,
        text_content,
        html_content,
        idempotency_key,
    } = form.0;
    let idempotency_key: IdempotencyKey = idempotency_key.try_into().map_err(e400)?;
    // Return early if we have a saved response in the database
    if let Some(saved_response) = get_saved_response(&pool, &idempotency_key, *user_id)
        .await
        .map_err(e500)?
    {
        return Ok(saved_response);
    }
    // [...]
}
```

11.8.3 Save Responses

We have code to retrieve saved responses, but we don't have code yet to save responses - that's what we will be focusing on next.

Let's add a new function skeleton to our `idempotency` module:

```rust
//! src/idempotency/mod.rs
// [...]
pub use persistence::save_response;
```

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn save_response(
    _pool: &PgPool,
    _idempotency_key: &IdempotencyKey,
    _user_id: Uuid,
    _http_response: &HttpResponse
) -> Result<(), anyhow::Error> {
    todo!()
}
```

We need to break `HttpResponse` into its separate components before we write the `INSERT` query. We can use `.status()` for the status code, `.headers()` for the headers... what about the body? There is a `.body()` method - this is its signature:

```rust
/// Returns a reference to this response's body.
pub fn body(&self) -> &B {
    self.res.body()
}
```

What is `B`? We must include the `impl` block definition into the picture to grasp it:

```rust
impl<B> HttpResponse<B> {
    /// Returns a reference to this response's body.
    pub fn body(&self) -> &B {
        self.res.body()
    }
}
```

Well, well, it turns out that `HttpResponse` is generic over the body type!
But, you may ask, "we have been using `HttpResponse` for 400 pages without specifying any generic parameter, what's going on?"

There is a default generic parameter which kicks in if `B` is left unspecified:

```rust
/// An outgoing response.
pub struct HttpResponse<B = BoxBody> {/* */}
```

11.8.3.1 `MessageBody` and HTTP Streaming

Why does `HttpResponse` need to be generic over the body type in the first place? Can't it just use `Vec<u8>` or a similar bytes container?

We have always worked with responses that were fully formed on the server before being sent back to the caller. HTTP/1.1 supports another mechanism to transfer data - `Transfer-Encoding: chunked`, also known as **HTTP streaming**.
The server breaks down the payload into multiple chunks and sends them over to the caller one at a time instead of accumulating the entire body in memory first. It allows the server to significantly reduce its memory usage. It is quite useful when working on large payloads such as files or results from a large query (streaming all the way through!).

With HTTP streaming in mind, it becomes easier to understand the design of `MessageBody`, the trait that must be implemented to use a type as body in `actix-web`:

```rust
pub trait MessageBody {
    type Error: Into<Box<dyn Error + 'static, Global>>;
    fn size(&self) -> BodySize;
    fn poll_next(
        self: Pin<&mut Self>,
        cx: &mut Context<'_>
    ) -> Poll<Option<Result<Bytes, Self::Error>>>;
    // [...]
}
```

You pull data, one chunk at a time, until you have fetched it all.
When the response is not being streamed, the data is available all at once - `poll_next` returns it all in one go.

Let's try to understand `BoxBody`, the default body type used by `HttpResponse`. The body type we have been using for several chapters, unknowingly!
`BoxBody` abstracts away the specific payload delivery mechanism. Under the hood, it is nothing more than an `enum` with a variant for each strategy, with a special case catering for body-less responses:

```rust
#[derive(Debug)]
pub struct BoxBody(BoxBodyInner);

enum BoxBodyInner {
    None(body::None),
    Bytes(Bytes),
    Stream(Pin<Box<dyn MessageBody<Error = Box<dyn StdError>>>>),
}
```

It worked for so long because we did not really care about the way the response was being sent back to the caller.
Implementing `save_response` forces us to look closer - we need to collect the response in memory[12] in order to save it in the `idempotency` table of our database.

`actix-web` has a dedicated function for situation like ours: `to_bytes`.
It calls `poll_next` until there is no more data to fetch, than it returns the entire response back to us inside a `Bytes` container[13].
I'd normally advise for caution when talking about `to_bytes` - if you are dealing with huge payloads, there is a risk of putting the server under significant memory pressure.
This is not our case - all our response bodies are small and don't actually take advantage of HTTP streaming, so `to_bytes` will not actually do any work.

Enough with the theory - let's piece it together:

```
//! src/idempotency/persistence.rs
use actix_web::body::to_bytes;
// [...]

pub async fn save_response(
    pool: &PgPool,
    idempotency_key: &IdempotencyKey,
    user_id: Uuid,
    http_response: &HttpResponse,
) -> Result<(), anyhow::Error> {
    let status_code = http_response.status().as_u16() as i16;
    let headers = {
        let mut h = Vec::with_capacity(http_response.headers().len());
        for (name, value) in http_response.headers().iter() {
            let name = name.as_str().to_owned();
            let value = value.as_bytes().to_owned();
            h.push(HeaderPairRecord { name, value });
        }
        h
    };
    let body = to_bytes(http_response.body()).await.unwrap();
    todo!()
}
```

The compiler is not happy:

```
error[E0277]: the trait bound `&BoxBody: MessageBody` is not satisfied
--> src/idempotency/persistence.rs
  |
  |     let body = to_bytes(http_response.body()).await.unwrap();
  |                -------- ^^^^^^^^^^^^^^^^^^^^
```

[12] We technically have another option: stream the response body directly to the database and then stream it back from the database directly to the caller.

[13] You can think of `Bytes` as a `Vec<u8>` with extra perks - check out the documentation of the `bytes` crate for more details.

```
the trait `MessageBody` is not implemented for `&BoxBody`
 |                |
 |                required by a bound introduced by this call
 |
 = help: the following implementations were found:
          <BoxBody as MessageBody>
```

`BoxBody` implements `MessageBody`, but `&BoxBody` doesn't - and `.body()` returns a reference, it does not give us ownership over the body.
Why do we need ownership? It's because of HTTP streaming, once again!

Pulling a chunk of data from the payload stream requires a mutable reference to the stream itself - once the chunk has been read, there is no way to "replay" the stream and read it again.

There is a common pattern to work around this:

- Get ownership of the body via `.into_parts()`;
- Buffer the whole body in memory via `to_bytes`;
- *Do whatever you have to do with the body*;
- Re-assemble the response using `.set_body()` on the request head.

`.into_parts()` requires ownership of `HttpResponse` - we'll have to change the signature of `save_response` to accommodate it. Instead of asking for a reference, we'll take ownership of the response and then return another owned `HttpResponse` in case of success.

Let's go for it:

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn save_response(
    // [...]
    // No longer a reference!
    http_response: HttpResponse,
) -> Result<HttpResponse, anyhow::Error> {
    let (response_head, body) = http_response.into_parts();
    // `MessageBody::Error` is not `Send` + `Sync`,
    // therefore it doesn't play nicely with `anyhow`
    let body = to_bytes(body).await.map_err(|e| anyhow::anyhow!("{}", e))?;
    let status_code = response_head.status().as_u16() as i16;
    let headers = {
        let mut h = Vec::with_capacity(response_head.headers().len());
        for (name, value) in response_head.headers().iter() {
            let name = name.as_str().to_owned();
            let value = value.as_bytes().to_owned();
            h.push(HeaderPairRecord { name, value });
        }
        h
    };

    // TODO: SQL query

    // We need `.map_into_boxed_body` to go from
    // `HttpResponse<Bytes>` to `HttpResponse<BoxBody>`
    let http_response = response_head.set_body(body).map_into_boxed_body();
    Ok(http_response)
}
```

That should compile, although it isn't particularly useful (yet).

11.8.3.2 Array Of Composite Postgres Types

Let's add the insertion query:

```
//! src/idempotency/persistence.rs
// [...]

pub async fn save_response(
    // [...]
) -> Result<HttpResponse, anyhow::Error> {
    // [...]
    sqlx::query!(
        r#"
        INSERT INTO idempotency (
            user_id,
            idempotency_key,
            response_status_code,
            response_headers,
            response_body,
            created_at
        )
        VALUES ($1, $2, $3, $4, $5, now())
        "#,
        user_id,
        idempotency_key.as_ref(),
        status_code,
        headers,
        body.as_ref()
    )
    .execute(pool)
    .await?;

    let http_response = response_head.set_body(body).map_into_boxed_body();
    Ok(http_response)
}
```

Compilation fails with an error:

```
error: unsupported type _header_pair for param #4
--> src/idempotency/persistence.rs
 |
 | /     sqlx::query!(
 | |         r#"
 | |         INSERT INTO idempotency (
 | |             user_id,
.. |
 | |         body.as_ref()
 | |     )
   | |_____^
```

It does make sense - we are using a custom type and `sqlx::query!` is not powerful enough to learn about it at compile-time in order to check our query. We will have to disable compile-time verification - use `query_unchecked!` instead of `query!`:

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn save_response(
    // [...]
) -> Result<HttpResponse, anyhow::Error> {
    // [...]
    sqlx::query_unchecked!(/* */)
    // [...]
}
```

We are getting closer - a different error!

```
error[E0277]: the trait bound `HeaderPairRecord: PgHasArrayType` is not satisfied
--> src/idempotency/persistence.rs
  |
  | /     sqlx::query_unchecked!(
  | |         r#"
  | |         INSERT INTO idempotency (
  | |             user_id,
..  |
  | |         body.as_ref()
  | |     )
  | |_____^ the trait `PgHasArrayType` is not implemented for `HeaderPairRecord`
```

`sqlx` knows, via our `#[sqlx(type_name = "header_pair")]` attribute, the name of the composite type itself. It does not know the name of the type for *arrays* containing `header_pair` elements. Postgres creates an array type implicitly when we run a `CREATE TYPE` statement - it is simply the composite type name prefixed by an underscore[14].

We can provide this information to `sqlx` by implementing the `PgHasArrayType` trait, just like the compiler suggested:

```rust
//! src/idempotency/persistence.rs
use sqlx::postgres::PgHasArrayType;
// [...]

impl PgHasArrayType for HeaderPairRecord {
    fn array_type_info() -> sqlx::postgres::PgTypeInfo {
        sqlx::postgres::PgTypeInfo::with_name("_header_pair")
    }
}
```

The code should finally compile.

11.8.3.3 Plug It In

It's a milestone, but it is a bit early to cheer - we don't know if it works yet. Our integration test is still red.
Let's plug `save_response` into our request handler:

```rust
//! src/routes/admin/newsletter/post.rs
use crate::idempotency::save_response;
```

[14] If the type name ends up being too long, some truncation takes place as well.

```rust
// [...]

pub async fn publish_newsletter(/* */) -> Result</* */> {
    // [...]
    for subscriber in subscribers {
        // [...]
    }
    FlashMessage::info("The newsletter issue has been published!").send();
    let response = see_other("/admin/newsletters");
    let response = save_response(&pool, &idempotency_key, *user_id, response)
        .await
        .map_err(e500)?;
    Ok(response)
}
```

Low and behold, `cargo test` succeeds! We made it!

11.9 Concurrent Requests

We dealt with the "easy" scenario when it comes to idempotency: a request arrives, it's fully processed, then a retry comes in.

We will now deal with the more troublesome scenario - the retry arrives before the first request is fully processed.
We expect the second request to be queued behind the first one - once that finishes, it will retrieve the saved HTTP response from the store and return it to the caller.

11.9.1 Requirements As Tests #2

We can, once again, encode our requirements as tests:

```rust
//! tests/api/newsletter.rs
use std::time::Duration;
// [...]

#[tokio::test]
async fn concurrent_form_submission_is_handled_gracefully() {
    // Arrange
    let app = spawn_app().await;
    create_confirmed_subscriber(&app).await;
    app.test_user.login(&app).await;

    Mock::given(path("/email"))
        .and(method("POST"))
        // Setting a long delay to ensure that the second request
        // arrives before the first one completes
        .respond_with(ResponseTemplate::new(200).set_delay(Duration::from_secs(2)))
        .expect(1)
        .mount(&app.email_server)
        .await;

    // Act - Submit two newsletter forms concurrently
    let newsletter_request_body = serde_json::json!({
        "title": "Newsletter title",
        "text_content": "Newsletter body as plain text",
        "html_content": "<p>Newsletter body as HTML</p>",
        "idempotency_key": uuid::Uuid::new_v4().to_string()
    });
    let response1 = app.post_publish_newsletter(&newsletter_request_body);
    let response2 = app.post_publish_newsletter(&newsletter_request_body);
    let (response1, response2) = tokio::join!(response1, response2);

    assert_eq!(response1.status(), response2.status());
    assert_eq!(response1.text().await.unwrap(), response2.text().await.unwrap());

    // Mock verifies on Drop that we have sent the newsletter email **once**
}
```

The test fails - our server returned a `500 Internal Server Error` to one of the two requests:

```
thread 'newsletter::concurrent_form_submission_is_handled_gracefully'
panicked at 'assertion failed: `(left == right)`
  left: `303`,
 right: `500`'
```

The logs explain what happened:

```
exception.details:
    error returned from database:
    duplicate key value violates unique constraint "idempotency_pkey"

    Caused by:
        duplicate key value violates unique constraint "idempotency_pkey"
```

The slowest request fails to insert into the `idempotency` table due to our uniqueness constraint.
The error response is not the only issue: both requests executed the email dispatch code (otherwise we wouldn't have seen the constraint violation!), resulting into duplicate delivery.

11.9.2 Synchronization

The second request is not aware of the first until it tries to insert into the database.
If we want to prevent duplicate delivery, we need to introduce **cross-request synchronization** *before* we start processing subscribers.

In-memory locks (e.g. `tokio::sync::Mutex`) would work if all incoming requests were being served by a single API instance. This is not our case: our API is replicated, therefore the two requests might end up being processed by two different instances.

Our synchronization mechanism will have to live out-of-process - our database being the natural candidate.
Let's think about it: we have an `idempotency` table, it contains one row for each unique combination of user id and idempotency key. Can we do something with it?

Our current implementation inserts a row into the `idempotency` table *after* processing the request, just before returning the response to the caller. We are going to change that: we will insert a new row as soon as the handler is invoked.
We don't know the final response at that point - we haven't started processing yet! We must relax the `NOT NULL` constraints on some of the columns:

```
sqlx migrate add relax_null_checks_on_idempotency
```

```sql
ALTER TABLE idempotency ALTER COLUMN response_status_code DROP NOT NULL;
ALTER TABLE idempotency ALTER COLUMN response_body DROP NOT NULL;
ALTER TABLE idempotency ALTER COLUMN response_headers DROP NOT NULL;
```

```
sqlx migrate run
```

We can now insert a row as soon as the handler gets invoked using the information we have up to that point - the user id and the idempotency key, our composite primary key.

The first request will succeed in inserting a row into `idempotency`. The second request, instead, will fail due to our uniqueness constraint.
That is not what we want:

- if the first request completed, we want to return the saved response;
- if the first request is still ongoing, we want to **wait**.

The first scenario can be accommodated by using Postgres' `ON CONFLICT` statement - it allows us to define what should happen when an `INSERT` fails due to a constraint violation (e.g. uniqueness). We have two options: `ON CONFLICT DO NOTHING` and `ON CONFLICT DO UPDATE`.
`ON CONFLICT DO NOTHING`, as you might guess, does nothing - it simply swallows the error. We can detect that the row was already there by checking the number of rows that were affected by the statement.
`ON CONFLICT DO UPDATE`, instead, can be used to modify the pre-existing row - e.g. `ON CONFLICT DO UPDATE SET updated_at = now()`.

We will use `ON CONFLICT DO NOTHING` - if no new row was inserted, we will try to fetch the saved response.
Before we start implementing, there is an issue we need to solve: our code no longer compiles. Our code has not been updated to deal with the fact that a few columns in `idempotency` are now nullable. We must update the query to ask `sqlx` to forcefully assume that the columns will not be null - if we are wrong, it will cause an error at runtime.
The syntax is similar to the type casting syntax we used previously to deal with header pairs - we must append a `!` to the column alias name:

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn get_saved_response(/* */) -> Result</* */> {
    let saved_response = sqlx::query!(
        r#"
        SELECT
            response_status_code as "response_status_code!",
            response_headers as "response_headers!: Vec<HeaderPairRecord>",
            response_body as "response_body!"
        [...]
        "#,
        user_id,
        idempotency_key.as_ref()
    )
    // [...]
}
```

Let's now define the skeleton of a new function, the one we will invoke at the beginning of our request handler - `try_processing`.
It will try to perform the insertion we just discussed - if it fails because a row already exists, we will assume that a response has been saved and try to return it.

```rust
//! src/idempotency/mod.rs
// [...]
pub use persistence::{try_processing, NextAction};
```

```rust
//! src/idempotency/persistence.rs
// [...]

pub enum NextAction {
    StartProcessing,
    ReturnSavedResponse(HttpResponse)
}

pub async fn try_processing(
    pool: &PgPool,
    idempotency_key: &IdempotencyKey,
    user_id: Uuid
) -> Result<NextAction, anyhow::Error> {
    todo!()
}
```

Our handler will invoke `try_processing` instead of `get_saved_response`:

```rust
//! src/routes/admin/newsletter/post.rs
use crate::idempotency::{try_processing, NextAction};
// [...]

pub async fn publish_newsletter(/* */) -> Result<HttpResponse, actix_web::Error> {
    // [...]
    let idempotency_key: IdempotencyKey = idempotency_key.try_into().map_err(e400)?;
    match try_processing(&pool, &idempotency_key, *user_id)
        .await
        .map_err(e500)?
    {
        NextAction::StartProcessing => {}
        NextAction::ReturnSavedResponse(saved_response) => {
            success_message().send();
            return Ok(saved_response);
        }
    }
    // [...]
    success_message().send();
    let response = see_other("/admin/newsletters");
    let response = save_response(&pool, &idempotency_key, *user_id, response)
        .await
        .map_err(e500)?;
    Ok(response)
}

fn success_message() -> FlashMessage {
    FlashMessage::info("The newsletter issue has been published!")
}
```

We can now flesh out `try_processing`:

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn try_processing(
    pool: &PgPool,
    idempotency_key: &IdempotencyKey,
    user_id: Uuid,
) -> Result<NextAction, anyhow::Error> {
    let n_inserted_rows = sqlx::query!(
        r#"
        INSERT INTO idempotency (
            user_id,
            idempotency_key,
            created_at
        )
        VALUES ($1, $2, now())
        ON CONFLICT DO NOTHING
        "#,
        user_id,
        idempotency_key.as_ref()
    )
    .execute(pool)
    .await?
    .rows_affected();
    if n_inserted_rows > 0 {
        Ok(NextAction::StartProcessing)
    } else {
        let saved_response = get_saved_response(pool, idempotency_key, user_id)
            .await?
            .ok_or_else(||
                anyhow::anyhow!("We expected a saved response, we didn't find it")
            )?;
        Ok(NextAction::ReturnSavedResponse(saved_response))
    }
}
```

A bunch of our tests will start failing. What is going on?
Log inspection highlights a `duplicate key value violates unique constraint "idempotency_pkey"`. Guess what? We forgot to update `save_response`! It's trying to insert *another* row into `idempotency` for the same combination of user id and idempotency key - it needs to perform an `UPDATE` instead of an `INSERT`.

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn save_response(/* */) -> Result</* */> {
    // [...]
    sqlx::query_unchecked!(
        r#"
        UPDATE idempotency
        SET
            response_status_code = $3,
            response_headers = $4,
            response_body = $5
        WHERE
```

```
            user_id = $1 AND
            idempotency_key = $2
        "#,
        user_id,
        idempotency_key.as_ref(),
        status_code,
        headers,
        body.as_ref()
    )
    // [...]
}
```

We are back to square one - `concurrent_form_submission_is_handled_gracefully` is the only failing test. What have we gained?
Very little - the second request returns an error instead of sending emails out twice. An improvement, but not yet where we want to land.

We need to find a way to cause the `INSERT` in `try_processing` to wait instead of erroring out when a retry arrives before the first request has completed processing.

11.9.2.1 Transaction Isolation Levels

Let's do an experiment: we will wrap the `INSERT` in `try_processing` and the `UPDATE` in `save_response` in a single SQL transaction.
What do you think it's going to happen?

```rust
//! src/idempotency/persistence.rs
use sqlx::{Postgres, Transaction};
// [...]

#[allow(clippy::large_enum_variant)]
pub enum NextAction {
    // Return transaction for later usage
    StartProcessing(Transaction<'static, Postgres>),
    // [...]
}


pub async fn try_processing(/* */) -> Result</* */> {
    let mut transaction = pool.begin().await?;
    let n_inserted_rows = sqlx::query!(/* */)
        .execute(&mut transaction)
        .await?
        .rows_affected();
    if n_inserted_rows > 0 {
        Ok(NextAction::StartProcessing(transaction))
    } else {
        // [...]
    }
}

pub async fn save_response(
    // No longer a `Pool`!
    mut transaction: Transaction<'static, Postgres>,
    // [...]
) -> Result</* */> {
    // [...]
    sqlx::query_unchecked!(/* */)
        .execute(&mut transaction)
        .await?;
    transaction.commit().await?;
    // [...]
}
```

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

pub async fn publish_newsletter(/* */) -> Result</* */> {
    // [...]
    let transaction = match try_processing(&pool, &idempotency_key, *user_id)
        .await
        .map_err(e500)?
    {
        NextAction::StartProcessing(t) => t,
        // [...]
    };
    // [...]
    let response = save_response(transaction, /* */)
        .await
        .map_err(e500)?;
    // [...]
}
```

All our tests are passing! But **why**?
It boils down to locks and transaction isolation levels!

`READ COMMITTED` is the default isolation level in Postgres. We have not tuned this setting, therefore this is the case for the queries in our application as well.
Postgres' documentation describes the behaviour at this isolation level as follows:

> [...] a SELECT query (without a FOR UPDATE/SHARE clause) sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions. In effect, a SELECT query sees a snapshot of the database as of the instant the query begins to run.

Data-altering statements, instead, will be influenced by uncommitted transactions that are trying to alter the same set of rows:

> `UPDATE`, `DELETE`, `SELECT FOR UPDATE` [...] will only find target rows that were committed as of the command start time. However, such a target row might have already been updated (or deleted or locked) by another concurrent transaction by the time it is found. In this case, **the would-be updater will wait for the first updating transaction to commit or roll back (if it is still in progress)**.

This is exactly what is happening in our case.
The `INSERT` statement fired by the second request must wait for outcome of the SQL transaction started by the first request.

If the latter commits, the former will `DO NOTHING`.
If the latter rolls back, the former will actually perform the insertion.

It is worth highlighting that this strategy will **not** work if using stricter isolation levels.
We can test this pretty easily:

```rust
//! src/idempotency/persistence.rs
// [...]

pub async fn try_processing(/* */) -> Result</* */> {
    let mut transaction = pool.begin().await?;
    sqlx::query!("SET TRANSACTION ISOLATION LEVEL repeatable read")
        .execute(&mut transaction)
        .await?;
    let n_inserted_rows = sqlx::query!(/* */)
    // [...]
}
```

The second concurrent request will fail due to a database error: `could not serialize access due to concurrent update`.
`repeatable read` is designed to prevent non-repeatable reads (who would have guessed?): the same `SELECT` query, if run twice in a row within the same transaction, should return the same data.
This has consequences for statements such as `UPDATE`: if they are executed within a `repeatable read` transaction, they cannot modify or lock rows changed by other transactions after the repeatable read transaction began.
This is why the transaction initiated by the second request fails to commit in our little experiment above. The same would have happened if we had chosen `serializable`, the strictest isolation level available in Postgres.

11.10 Dealing With Errors

We made some solid progress - our implementation handles duplicated requests gracefully, no matter if they arrive concurrently or sequentially.
What about errors?

Let's add another test case:

```toml
#! Cargo.toml
# [...]
[dev-dependencies]
serde_urlencoded = "0.7.1"
# [...]
```

```rust
//! tests/api/newsletter.rs
use fake::faker::internet::en::SafeEmail;
use fake::faker::name::en::Name;
use fake::Fake;
use wiremock::MockBuilder;
// [...]

// Short-hand for a common mocking setup
fn when_sending_an_email() -> MockBuilder {
    Mock::given(path("/email")).and(method("POST"))
}

#[tokio::test]
async fn transient_errors_do_not_cause_duplicate_deliveries_on_retries() {
    // Arrange
    let app = spawn_app().await;
    let newsletter_request_body = serde_json::json!({
        "title": "Newsletter title",
        "text_content": "Newsletter body as plain text",
        "html_content": "<p>Newsletter body as HTML</p>",
        "idempotency_key": uuid::Uuid::new_v4().to_string()
    });
    // Two subscribers instead of one!
    create_confirmed_subscriber(&app).await;
    create_confirmed_subscriber(&app).await;
    app.test_user.login(&app).await;

    // Part 1 - Submit newsletter form
    // Email delivery fails for the second subscriber
    when_sending_an_email()
        .respond_with(ResponseTemplate::new(200))
        .up_to_n_times(1)
        .expect(1)
        .mount(&app.email_server)
        .await;
    when_sending_an_email()
        .respond_with(ResponseTemplate::new(500))
        .up_to_n_times(1)
        .expect(1)
        .mount(&app.email_server)
        .await;

    let response = app.post_publish_newsletter(&newsletter_request_body).await;
    assert_eq!(response.status().as_u16(), 500);

    // Part 2 - Retry submitting the form
    // Email delivery will succeed for both subscribers now
    when_sending_an_email()
        .respond_with(ResponseTemplate::new(200))
        .expect(1)
        .named("Delivery retry")
        .mount(&app.email_server)
        .await;
    let response = app.post_publish_newsletter(&newsletter_request_body).await;
    assert_eq!(response.status().as_u16(), 303);

    // Mock verifies on Drop that we did not send out duplicates
}

async fn create_unconfirmed_subscriber(app: &TestApp) -> ConfirmationLinks {
    // We are working with multiple subscribers now,
    // their details must be randomised to avoid conflicts!
    let name: String = Name().fake();
    let email: String = SafeEmail().fake();
    let body = serde_urlencoded::to_string(&serde_json::json!({
        "name": name,
        "email": email
    }))
    .unwrap();
    // [...]
}
```

The test does not pass - we are seeing yet another instance of duplicated delivery:

```
thread 'newsletter::transient_errors_do_not_cause_duplicate_deliveries_on_retries'
panicked at 'Verifications failed:
- Delivery retry.
        Expected range of matching incoming requests: == 1
        Number of matched incoming requests: 2
```

It makes sense, if you think again about our idempotency implementation: the SQL transaction inserting into the `idempotency` table commits exclusively when processing succeeds.
Errors lead to an early return - this triggers a rollback when the `Transaction<'static, Postgres>` value is dropped.

Can we do better?

11.10.1 Distributed Transactions

The pain we are feeling is a common issue in real-world applications - you lose transactionality[15] when executing logic that touches, at the same time, your local state and a remote state managed by another system[16].

I happen to be fascinated by the technical challenges in distributed systems, but I am well aware that users do not share my passion. They want to get something done, they do not care about the internals - and rightly so.
A newsletter author expects one of the following scenarios after clicking on `Submit`:

- the issue was delivered to all subscribers;
- the issue could not be published, therefore nobody received it.

Our implementation allows for a third scenario at the moment: the issue could not be published (`500 Internal Server Error`), but *some* subscribers received it anyway.
That won't do - **partial** execution is not acceptable, the system must end up in a sensible state.
There are two common approaches to solve this problem: **backward recovery** and **forward recovery**.

11.10.2 Backward Recovery

Backward recovery tries to achieve a semantic rollback by executing **compensating actions**.
Let's imagine we are working on an e-commerce checkout system: we have already charged the customer for the products in their basket but, when trying to authorize the shipment, we discover

[15] Protocols like 2-phase commit would allow us to have an all-or-nothing semantic in a distributed system, but they are not widely supported due to their complexity and drawbacks.

[16] More often than not, the other system lives within your organization - it's just a different microservice, with its own isolated data store. You have traded the inner complexity of the monolith for the complexity of orchestrating changes across multiple sub-system - complexity has to live somewhere.

that one of the items is now out of stock.
We can perform a backward recovery by cancelling all shipment instructions and refunding the customer for the entire amount of their basket.
The recovery mechanism is not transparent to the customer - they will still see two payments on their transaction history, the original charge and the following refund. We are also likely to send out an email to explain what happened. But their balance, the *state* they care about, has been restored.

11.10.3 Forward Recovery

Backward recovery is not a good fit for our newsletter delivery system - we cannot "unsend" an email nor would it make sense to send a follow-up email asking subscribers to ignore the email we sent before (it'd be funny though).

We must try to perform **forward recovery** - drive the overall workflow to completion even if one or more sub-tasks did not succeed.
We have two options: **active** and **passive** recovery.

Passive recovery pushes on the API caller the responsibility to drive the workflow to completion. The request handler leverages *checkpoints* to keep track of its progress - e.g. "123 emails have been sent out". If the handler crashes, the next API call will resume processing from the latest checkpoint, minimizing the amount of duplicated work (if any). After enough retries, the workflow will eventually complete.

Active recovery, instead, does not require the caller to do anything apart from kicking off the workflow. The system must self-heal.
We would rely on a background process - e.g. a background task on our API - to detect newsletter issues whose delivery stopped halfway. The process would then drive the delivery to completion.
Healing would happen **asynchronously** - outside the lifecycle of the original `POST /admin/newsletters` request.

Passive recovery makes for a poor user experience - the newsletter author has to submit the form over and over again until they receive a success response back. The author is in an awkward position - is the error they are seeing related to a transient issue encountered during delivery? Or is it the database failing when trying to fetch the list of subscribers? In other words, will retries actually lead to a success, eventually?
If they choose to give up retrying, while in the middle of delivery, the system is once again left in an inconsistent state.

We will therefore opt for active recovery in our implementation.

11.10.4 Asynchronous Processing

Active recovery has its rough edges as well.
We do not want the author to receive an error back from the API while, under the hood, newsletter delivery has been kicked off.

We can improve the user experience by changing the expectations for `POST /admin/newsletters`. A successful form submission currently implies that the new newsletter issue has been validated and delivered to all subscribers.

We can reduce its scope: a successful form submission will mean that the newsletter has been validated and **will** be delivered to all subscribers, **asynchronously**.

In other words, a successful form submission guarantees to the author that the delivery workflow has been correctly kicked off. They just need to wait for all emails to go out, but they have nothing to worry about - it will happen[17].

The request handler of `POST /admin/newsletters` is no longer going to dispatch emails - it will simply enqueue a list of tasks that will be fulfilled asynchronously by a set of background workers. We will use another Postgres table as our task queue - it will be named `issue_delivery_queue`.

At a glance, it might look like a small difference - we are just shifting around when work needs to happen. But it has a powerful implication: we recover transactionality.
Our subscribers' data, our idempotency records, the task queue - they all live in Postgres. All the operations performed by `POST /admin/newsletters` can be wrapped in a single SQL transaction - either they all succeed, or nothing happened.
The caller no longer needs to second guess the response of our API or try to reason about its implementation!

11.10.4.1 `newsletter_issues`

By dispatching eagerly, we never needed to store the details of the issues we were sending out. To pursue our new strategy, this has to change: we will start persisting newsletter issues in a dedicated `newsletter_issues` table.
The schema should not come as a surprise:

```
sqlx migrate add create_newsletter_issues_table
```

```sql
-- migrations/20220211080603_create_newsletter_issues_table.sql
CREATE TABLE newsletter_issues (
   newsletter_issue_id uuid NOT NULL,
   title TEXT NOT NULL,
   text_content TEXT NOT NULL,
   html_content TEXT NOT NULL,
   published_at TEXT NOT NULL,
   PRIMARY KEY(newsletter_issue_id)
);
```

```
sqlx migrate run
```

Let's write a matching `insert_newsletter_issue` function - we'll need it soon:

```rust
//! src/routes/admin/newsletter/post.rs
use sqlx::{Postgres, Transaction};
use uuid::Uuid;
// [...]

#[tracing::instrument(skip_all)]
async fn insert_newsletter_issue(
    transaction: &mut Transaction<'_, Postgres>,
    title: &str,
```

[17] The author would still benefit from having visibility into the delivery process - e.g. a page to track how many emails are still outstanding for a certain newsletter issue. Workflow observability is out of scope for the book, but it might be an interesting exercise to pursue on your own.

```rust
    text_content: &str,
    html_content: &str,
) -> Result<Uuid, sqlx::Error> {
    let newsletter_issue_id = Uuid::new_v4();
    sqlx::query!(
        r#"
        INSERT INTO newsletter_issues (
            newsletter_issue_id,
            title,
            text_content,
            html_content,
            published_at
        )
        VALUES ($1, $2, $3, $4, now())
        "#,
        newsletter_issue_id,
        title,
        text_content,
        html_content
    )
    .execute(transaction)
    .await?;
    Ok(newsletter_issue_id)
}
```

11.10.4.2 `issue_delivery_queue`

When it comes to tasks, we are going to keep it simple:

```
sqlx migrate add create_issue_delivery_queue_table
```

```sql
-- migrations/20220211080603_create_issue_delivery_queue_table.sql
CREATE TABLE issue_delivery_queue (
   newsletter_issue_id uuid NOT NULL REFERENCES newsletter_issues (newsletter_issue_id),
   subscriber_email TEXT NOT NULL,
   PRIMARY KEY(newsletter_issue_id, subscriber_email)
);
```

```
sqlx migrate run
```

We can create the task set using a single insert query:

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

#[tracing::instrument(skip_all)]
async fn enqueue_delivery_tasks(
    transaction: &mut Transaction<'_, Postgres>,
    newsletter_issue_id: Uuid,
) -> Result<(), sqlx::Error> {
    sqlx::query!(
        r#"
        INSERT INTO issue_delivery_queue (
            newsletter_issue_id,
            subscriber_email
        )
        SELECT $1, email
        FROM subscriptions
        WHERE status = 'confirmed'
        "#,
        newsletter_issue_id,
    )
    .execute(transaction)
    .await?;
    Ok(())
}
```

11.10.4.3 `POST /admin/newsletters`

We are ready to overhaul our request handler by putting together the pieces we just built:

```rust
//! src/routes/admin/newsletter/post.rs
// [...]

#[tracing::instrument(
    name = "Publish a newsletter issue",
    skip_all,
    fields(user_id=%&*user_id)
)]
pub async fn publish_newsletter(
    form: web::Form<FormData>,
    pool: web::Data<PgPool>,
    user_id: web::ReqData<UserId>,
) -> Result<HttpResponse, actix_web::Error> {
    let user_id = user_id.into_inner();
    let FormData {
        title,
        text_content,
        html_content,
        idempotency_key,
    } = form.0;
    let idempotency_key: IdempotencyKey = idempotency_key.try_into().map_err(e400)?;
    let mut transaction = match try_processing(&pool, &idempotency_key, *user_id)
        .await
        .map_err(e500)?
    {
        NextAction::StartProcessing(t) => t,
        NextAction::ReturnSavedResponse(saved_response) => {
            success_message().send();
            return Ok(saved_response);
        }
    };
    let issue_id = insert_newsletter_issue(&mut transaction, &title, &text_content, &html_content)
        .await
        .context("Failed to store newsletter issue details")
        .map_err(e500)?;
    enqueue_delivery_tasks(&mut transaction, issue_id)
        .await
        .context("Failed to enqueue delivery tasks")
        .map_err(e500)?;
    let response = see_other("/admin/newsletters");
    let response = save_response(transaction, &idempotency_key, *user_id, response)
        .await
        .map_err(e500)?;
    success_message().send();
    Ok(response)
}


fn success_message() -> FlashMessage {
    FlashMessage::info(
        "The newsletter issue has been accepted - \
        emails will go out shortly.",
    )
}
```

We can also delete `get_confirmed_subscribers` and `ConfirmedSubscriber`.

The logic in the request handler is now quite linear. The author is also going to have a quicker feedback loop - the endpoint no longer has to iterate over hundreds of subscribers before redirecting them to a success page.

11.10.4.4 Email Processing

Let's move our focus to the delivery instead.
We need to consume tasks from `issue_delivery_queue`. There are going to be multiple delivery workers running at the same time - at least one per API instance.
A naive approach would get us into trouble:

```sql
SELECT (newsletter_issue_id, subscriber_email)
FROM issue_delivery_queue
LIMIT 1
```

Multiple workers would pick the same task and we would end up with a lot of duplicated emails.

We need synchronization. Once again, we are going to leverage the database - we will use row-level locks.

Postgres 9.5 introduced the `SKIP LOCKED` clause - it allows `SELECT` statements to ignore all rows that are currently locked by another concurrent operation.
`FOR UPDATE`, instead, can be used to lock the rows returned by a `SELECT`.
We are going to combine them:

```sql
SELECT (newsletter_issue_id, subscriber_email)
FROM issue_delivery_queue
FOR UPDATE
SKIP LOCKED
LIMIT 1
```

This gives us a concurrency-safe queue.
Each worker is going to select an uncontested task (`SKIP LOCKED` and `LIMIT 1`); the task itself is going to become unavailable to other workers (`FOR UPDATE`) for the duration of the over-arching SQL transaction.
When the task is complete (i.e. the email has been sent), we are going to delete the corresponding row from `issue_delivery_queue` and commit our changes.

Let's code it up:

```
//! lib.rs
// [...]
pub mod issue_delivery_worker;
```

```
//! src/issue_delivery_worker;
use crate::email_client::EmailClient;
use sqlx::{PgPool, Postgres, Transaction};
use tracing::{field::display, Span};
use uuid::Uuid;

#[tracing::instrument(
    skip_all,
    fields(
        newsletter_issue_id=tracing::field::Empty,
        subscriber_email=tracing::field::Empty
    ),
    err
)]
async fn try_execute_task(
    pool: &PgPool,
    email_client: &EmailClient
) -> Result<(), anyhow::Error> {
    if let Some((transaction, issue_id, email)) = dequeue_task(pool).await? {
        Span::current()
            .record("newsletter_issue_id", &display(issue_id))
            .record("subscriber_email", &display(&email));
        // TODO: send email
        delete_task(transaction, issue_id, &email).await?;
    }
    Ok(())
}

type PgTransaction = Transaction<'static, Postgres>;

#[tracing::instrument(skip_all)]
async fn dequeue_task(
    pool: &PgPool,
) -> Result<Option<(PgTransaction, Uuid, String)>, anyhow::Error> {
    let mut transaction = pool.begin().await?;
    let r = sqlx::query!(
        r#"
        SELECT newsletter_issue_id, subscriber_email
        FROM issue_delivery_queue
        FOR UPDATE
        SKIP LOCKED
        LIMIT 1
        "#,
    )
    .fetch_optional(&mut transaction)
    .await?;
    if let Some(r) = r {
        Ok(Some((
            transaction,
            r.newsletter_issue_id,
            r.subscriber_email,
        )))
    } else {
```

```rust
        Ok(None)
    }
}

#[tracing::instrument(skip_all)]
async fn delete_task(
    mut transaction: PgTransaction,
    issue_id: Uuid,
    email: &str,
) -> Result<(), anyhow::Error> {
    sqlx::query!(
        r#"
        DELETE FROM issue_delivery_queue
        WHERE
            newsletter_issue_id = $1 AND
            subscriber_email = $2
        "#,
        issue_id,
        email
    )
    .execute(&mut transaction)
    .await?;
    transaction.commit().await?;
    Ok(())
}
```

To actually send the email, we need to fetch the newsletter content first:

```rust
//! src/issue_delivery_worker;
// [...]

struct NewsletterIssue {
    title: String,
    text_content: String,
    html_content: String,
}

#[tracing::instrument(skip_all)]
async fn get_issue(
    pool: &PgPool,
    issue_id: Uuid
) -> Result<NewsletterIssue, anyhow::Error> {
    let issue = sqlx::query_as!(
        NewsletterIssue,
        r#"
        SELECT title, text_content, html_content
        FROM newsletter_issues
        WHERE
            newsletter_issue_id = $1
        "#,
        issue_id
    )
    .fetch_one(pool)
    .await?;
    Ok(issue)
}
```

We can then recover the dispatch logic that used to live in `POST /admin/newsletters`:

```rust
//! src/issue_delivery_worker;
use crate::domain::SubscriberEmail;
// [...]

#[tracing::instrument(/* */)]
async fn try_execute_task(
    pool: &PgPool,
    email_client: &EmailClient
) -> Result<(), anyhow::Error> {
    if let Some((transaction, issue_id, email)) = dequeue_task(pool).await? {
        // [...]
        match SubscriberEmail::parse(email.clone()) {
            Ok(email) => {
                let issue = get_issue(pool, issue_id).await?;
                if let Err(e) = email_client
                    .send_email(
                        &email,
                        &issue.title,
                        &issue.html_content,
                        &issue.text_content,
                    )
                    .await
                {
                    tracing::error!(
                        error.cause_chain = ?e,
                        error.message = %e,
                        "Failed to deliver issue to a confirmed subscriber. \
                         Skipping.",
                    );
                }
            }
            Err(e) => {
                tracing::error!(
                    error.cause_chain = ?e,
                    error.message = %e,
                    "Skipping a confirmed subscriber. \
                     Their stored contact details are invalid",
                );
            }
        }
        delete_task(transaction, issue_id, &email).await?;
    }
    Ok(())
}
```

As you can see, we do not retry when the delivery attempt fails due to a Postmark error.
This could be changed by enhancing `issue_delivery_queue` - e.g. adding a `n_retries` and `execute_after` columns to keep track of how many attempts have already taken place and how long we should wait before trying again. Try implementing it as an exercise!

11.10.4.5 Worker loop

`try_execute_task` tries to deliver a single email - we need a background task that keeps pulling from `issue_delivery_queue` and fulfills tasks as they become available.

We can use an infinite loop:

```rust
//! src/issue_delivery_worker;
use std::time::Duration;
// [...]

async fn worker_loop(
    pool: PgPool,
    email_client: EmailClient
) -> Result<(), anyhow::Error> {
    loop {
        if try_execute_task(&pool, &email_client).await.is_err() {
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
    }
}
```

If we experience a transient failure[18], we need to sleep for a while to improve our future chances of success. This could be further refined by introducing an exponential backoff with jitter.

There is another scenario we need to keep in mind, apart from failure: `issue_delivery_queue` might be empty.
When that is the case, `try_execute_task` is going to be invoked *continuously*. That translates into an avalanche of unnecessary queries to the database.
We can mitigate this risk by changing the signature of `try_execute_task` - we need to know if it actually managed to dequeue something.

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

enum ExecutionOutcome {
    TaskCompleted,
    EmptyQueue,
}

#[tracing::instrument(/* */)]
async fn try_execute_task(/* */) -> Result<ExecutionOutcome, anyhow::Error> {
    let task = dequeue_task(pool).await?;
    if task.is_none() {
        return Ok(ExecutionOutcome::EmptyQueue);
    }
    let (transaction, issue_id, email) = task.unwrap();
    // [...]
    Ok(ExecutionOutcome::TaskCompleted)
}
```

`worker_loop` can now become smarter:

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

async fn worker_loop(/* */) -> Result</* */> {
    loop {
```

[18] Almost all errors returned by `try_execute_task` are transient in nature, except for invalid subscriber emails - sleeping is not going to fix those. Try refining the implementation to distinguish between transient and fatal failures, empowering `worker_loop` to react appropriately.

```
        match try_execute_task(&pool, &email_client).await {
            Ok(ExecutionOutcome::EmptyQueue) => {
                tokio::time::sleep(Duration::from_secs(10)).await;
            }
             Err(_) => {
                tokio::time::sleep(Duration::from_secs(1)).await;
            }
            Ok(ExecutionOutcome::TaskCompleted) => {}
        }
    }
}
```

No more busy looping, yay!

11.10.4.6 Launching Background Workers

We have a worker loop - but it is not launched anywhere.
Let's start by building the required dependencies based on the configuration values[19]:

```
//! src/issue_delivery_worker.rs
use crate::{configuration::Settings, startup::get_connection_pool};
// [...]

pub async fn run_worker_until_stopped(
    configuration: Settings
) -> Result<(), anyhow::Error> {
    let connection_pool = get_connection_pool(&configuration.database);

    let sender_email = configuration
        .email_client
        .sender()
        .expect("Invalid sender email address.");
    let timeout = configuration.email_client.timeout();
    let email_client = EmailClient::new(
        configuration.email_client.base_url,
        sender_email,
        configuration.email_client.authorization_token,
        timeout,
    );
    worker_loop(connection_pool, email_client).await
}
```

To run our background worker and the API side-to-side we need to restructure our `main` function. We are going to build the `Future` for each of the two long-running tasks - `Future`s are lazy in Rust, so nothing happens until they are actually awaited.
We will use `tokio::select!` to get both tasks to make progress concurrently. `tokio::select!` returns as soon as one of the two tasks completes or errors out:

[19] We are not re-using the dependencies we built for our `actix_web` application. This separation enables us, for example, to precisely control how many database connections are allocated to background tasks vs our API workloads. At the same time, this is clearly unnecessary at this stage: we could have built a single pool and HTTP client, passing `Arc` pointers to both sub-systems (API and worker). The right choice depends on the circumstances and the overall set of constraints.

```rust
//! src/main.rs
use zero2prod::issue_delivery_worker::run_worker_until_stopped;
// [...]

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    let subscriber = get_subscriber("zero2prod".into(), "info".into(), std::io::stdout);
    init_subscriber(subscriber);

    let configuration = get_configuration().expect("Failed to read configuration.");
    let application = Application::build(configuration.clone())
        .await?
        .run_until_stopped();
    let worker = run_worker_until_stopped(configuration);

    tokio::select! {
        _ = application => {},
        _ = worker => {},
    };

    Ok(())
}
```

There is a pitfall to be mindful of when using `tokio::select!` - all selected `Futures` are polled as a single task. This has consequences, as `tokio`'s documentation highlights:

> By running all async expressions on the current task, the expressions are able to run concurrently but not in parallel. This means all expressions are run on the same thread and if one branch blocks the thread, all other expressions will be unable to continue. If parallelism is required, spawn each async expression using `tokio::spawn` and pass the join handle to `select!`.

We should definitely follow their recommendation:

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

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // [...]
    let application = Application::build(configuration.clone()).await?;
    let application_task = tokio::spawn(application.run_until_stopped());
    let worker_task = tokio::spawn(run_worker_until_stopped(configuration));

    tokio::select! {
        _ = application_task => {},
        _ = worker_task => {},
    };

    Ok(())
}
```

As it stands, we have no visibility into which task completed first or if they completed successfully at all. Let's add some logging:

```rust
//! src/main.rs
use std::fmt::{Debug, Display};
use tokio::task::JoinError;
// [...]

#[tokio::main]
async fn main() -> anyhow::Result<()> {
    // [...]
    tokio::select! {
        o = application_task => report_exit("API", o),
        o = worker_task =>  report_exit("Background worker", o),
    };

    Ok(())
}


fn report_exit(
    task_name: &str,
    outcome: Result<Result<(), impl Debug + Display>, JoinError>
) {
    match outcome {
        Ok(Ok(())) => {
            tracing::info!("{} has exited", task_name)
        }
        Ok(Err(e)) => {
            tracing::error!(
                error.cause_chain = ?e,
                error.message = %e,
                "{} failed",
                task_name
            )
        }
        Err(e) => {
            tracing::error!(
                error.cause_chain = ?e,
                error.message = %e,
                "{}' task failed to complete",
                task_name
            )
        }
    }
}
```

It is looking pretty solid!

11.10.4.7 Updating The Test Suite

We have one little problem left - many of our tests are failing. They were written when emails were being delivered synchronously, which is no longer the case. How should we deal with them?

Launching a background worker would mimic the behaviour of our application, but it would make for a fragile test suite - we would have to sleep for arbitrary time intervals waiting for the background worker to process the email tasks we just enqueued. We are bound to have flaky tests sooner or later.

An alternative approach relies on launching the worker on demand, asking it to consume all available tasks. It deviates slightly from the behaviour in our `main` function, but it manages to exercise most of the code while being significantly more robust. This is what we are going for!

Let's add an `EmailClient` instance to our `TestApp`:

```rust
//! src/configuration.rs
use crate::email_client::EmailClient;
// [...]

impl EmailClientSettings {
    pub fn client(self) -> EmailClient {
        let sender_email = self.sender().expect("Invalid sender email address.");
        let timeout = self.timeout();
        EmailClient::new(
            self.base_url,
            sender_email,
            self.authorization_token,
            timeout,
        )
    }

    // [...]
}
```

```rust
//! tests/api/helpers.rs
use zero2prod::email_client::EmailClient;
// [...]

pub struct TestApp {
    // [...]
    pub email_client: EmailClient
}

pub async fn spawn_app() -> TestApp {
    // [...]
    let test_app = TestApp {
        // [...]
        email_client: configuration.email_client.client()
    };
    // [...]
}
```

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

pub async fn run_worker_until_stopped(
    configuration: Settings
) -> Result<(), anyhow::Error> {
    let connection_pool = get_connection_pool(&configuration.database);
    // Use helper function!
    let email_client = configuration.email_client.client();
    worker_loop(connection_pool, email_client).await
}
```

```rust
//! src/startup.rs
// [...]
impl Application {
    pub async fn build(configuration: Settings) -> Result<Self, anyhow::Error> {
        let connection_pool = get_connection_pool(&configuration.database);
        // Use helper function!
        let email_client = configuration.email_client.client();
        // [...]
    }
    // [...]
}
```

We can then write a helper to consume all enqueued tasks:

```rust
//! tests/api/helpers.rs
use zero2prod::issue_delivery_worker::{try_execute_task, ExecutionOutcome};
// [...]

impl TestApp {
    pub async fn dispatch_all_pending_emails(&self) {
        loop {
            if let ExecutionOutcome::EmptyQueue =
                try_execute_task(&self.db_pool, &self.email_client)
                    .await
                    .unwrap()
            {
                break;
            }
        }
    }
    // [...]
}
```

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

// Mark as pub
pub enum ExecutionOutcome {/* */}

#[tracing::instrument(/* */)]
// Mark as pub
pub async fn try_execute_task(/* */) -> Result</* */> {/* */}
```

We can update all the impacted test cases:

```rust
//! tests/api/newsletter.rs
// [...]

#[tokio::test]
async fn newsletters_are_not_delivered_to_unconfirmed_subscribers() {
    // [...]
    assert!(html_page.contains(
        "<p><i>The newsletter issue has been accepted - \
        emails will go out shortly.</i></p>"
    ));
    app.dispatch_all_pending_emails().await;
    // Mock verifies on Drop that we haven't sent the newsletter email
}

#[tokio::test]
```

```rust
async fn newsletters_are_delivered_to_confirmed_subscribers() {
    // [...]
    assert!(html_page.contains(
        "<p><i>The newsletter issue has been accepted - \
        emails will go out shortly.</i></p>"
    ));
    app.dispatch_all_pending_emails().await;
    // Mock verifies on Drop that we have sent the newsletter email
}

#[tokio::test]
async fn newsletter_creation_is_idempotent() {
    // [...]
    // Act - Part 2 - Follow the redirect
    let html_page = app.get_publish_newsletter_html().await;
    assert!(html_page.contains(
        "<p><i>The newsletter issue has been accepted - \
        emails will go out shortly.</i></p>"
    ));
    // [...]
    // Act - Part 4 - Follow the redirect
    let html_page = app.get_publish_newsletter_html().await;
    assert!(html_page.contains(
        "<p><i>The newsletter issue has been accepted - \
        emails will go out shortly.</i></p>"
    ));
    app.dispatch_all_pending_emails().await;
    // Mock verifies on Drop that we have sent the newsletter email **once**
}

#[tokio::test]
async fn concurrent_form_submission_is_handled_gracefully() {
    // [...]
    app.dispatch_all_pending_emails().await;
    // Mock verifies on Drop that we have sent the newsletter email **once**
}

// We deleted `transient_errors_do_not_cause_duplicate_deliveries_on_retries`
// It is no longer relevant given the redesign
```

The tests are passing, we made it!

Well, we *almost* made it.
We neglected one detail: there is no expiry mechanism for our idempotency keys. Try designing one as an exercise, using what we learned on background workers as a reference.

11.11 Epilogue

This is where our journey together comes to an end.

We started from an empty skeleton. Look at our project now: fully functional, well tested, reasonably secure - a proper minimum viable product. The project was never the goal though - it was an excuse, an opportunity to see what it feels like to write a production-ready API using Rust.

Zero To Production In Rust started with a question, a question I hear every other day:

> Can Rust be a productive language for API development?

I have taken you on a tour. I showed you a little corner of the Rust ecosystem, an opinionated yet powerful toolkit. I tried to explain, to the best of my abilities, the key language features.
The choice is now yours: you have learned enough to keep walking on your own, if you wish to do so.

Rust's adoption in the industry is taking off: we are living through an inflection point. It was my ambition to write a book that could serve as a ticket for this rising tide - an onboarding guide for those who want to be a part of this story.
This is just the beginning - the future of this community is yet to be written, but it is looking bright.