

Rust Servers, Services, and Apps

Prabhu Eshwarla

MEAP



MANNING

Rust Servers, Services, and Apps MEAP V13

1. [MEAP VERSION 13](#)
2. [Welcome](#)
3. [1 Why Rust for web applications?](#)
4. [2 Writing a basic web server from scratch](#)
5. [3 Building a RESTful Web Service](#)
6. [4 Performing database operations](#)
7. [5 Handling Errors](#)
8. [6 Evolving the APIs and fearless refactoring](#)
9. [7 Introduction to server-side web apps in Rust](#)
10. [8 Working with templates for tutor registration](#)
11. [9 Working with forms for course maintenance](#)
12. [10 Understanding Async Rust](#)
13. [11 Building a P2P node with Async Rust](#)
14. [12 Deploying web services with Docker](#)
15. [Appendix A. Postgres installation](#)

Rust Servers, Services, and Apps

Prabhu Eshwarla

MEAP

MANNING



MEAP VERSION 13

 MANNING PUBLICATIONS

Welcome

Thank you for purchasing the MEAP edition of *Rust Servers, Services, and Apps*.

Rust is a *hot* topic right now. It has been named *most loved programming language* in developer surveys for five consecutive years, and interest is growing among software developers and engineers alike, from both ends of the spectrum: low-level system programmers and higher-level application developers all want to explore and learn Rust. That said, one question that gets asked frequently is whether Rust is really suitable and ready for the web. Most of the learning material available in this space is really introductory in nature, and doesn't provide a view into how Rust can handle more complex scenarios encountered in web development. In this book, I aim to show you how Rust, in spite of its reputation for being a systems programming language, is really a surprisingly delightful language to build web applications in.

Of course, most (if not all) web development happens using web frameworks, rather than directly using a vanilla programming language. The web frameworks in Rust are much younger than full-featured and battle-tested frameworks like Rails, Django or Laravel. But in spite of its young ecosystem, Rust provides several compelling benefits for the web domain, including an expressive and static type system that translates to higher system reliability, lower and consistent resource usage, superior performance, and options for lower-level control than what is possible with other web development languages.

In this book I will show you how to apply Rust to the web domain. Using a practical full-length project, we will push the limits and see how Rust measures up to real-world challenges. We'll build a low-level web server, a web service, a server- rendered application, and a WASM-based front-end (single-page application), and these scenarios will give you a pretty good foundation to evaluate for yourself how you can apply Rust at work or to a side-project in the web domain. Perhaps more importantly, this book will

help you identify use cases for which you would not use Rust, and instead opt for the safety and comfort of another programming language and ecosystem. Along the way, I will also share practical tips and pitfalls, and best practices gained from my experience running Rust backend servers and applications in production environments.

If you have read *The Rust Programming Language* (a.k.a "the Book"), are eager to apply Rust to a practical domain that you are already familiar with, and are interested in strengthening your knowledge of Rust fundamentals, this book is for you. Further, I've made a conscious choice not to make this book all about learning any specific web framework or library (though I've made choices of tools for purposes of narrative and coding examples).

What I will not attempt to do in this book is to draw the battle-lines on which is the best programming language. So, if you are familiar with Rust but not yet convinced of its value proposition, this book may not be for you. Nor is this book aimed at people who have absolutely no knowledge of what Rust is about or what it offers. That said, if you've tried Rust and love it already, and are also interested in the web domain, I invite you to join me on the journey to explore Rust for the web, and I welcome your feedback in the [liveBook's Discussion Forum](#) for the book.

Best regards,

Prabhu Eshwarla

In this book

[MEAP VERSION 13](#) [About this MEAP](#) [Welcome](#) [Brief Table of Contents](#) [1 Why Rust for web applications?](#) [2 Writing a basic web server from scratch](#) [3 Building a RESTful Web Service](#) [4 Performing database operations](#) [5 Handling Errors](#) [6 Evolving the APIs and fearless refactoring](#) [7 Introduction to server-side web apps in Rust](#) [8 Working with templates for tutor registration](#) [9 Working with forms for course maintenance](#) [10 Understanding Async Rust](#) [11 Building a P2P node with Async Rust](#) [12 Deploying web services with Docker](#) [Appendix A. Postgres installation](#)

1 Why Rust for web applications?

This chapter covers

- Introduction to modern web applications
- Choosing Rust for web applications
- Visualizing the example application

Connected web applications that work over the internet form the backbone of modern businesses and human digital lives.

As individuals, we use consumer-focused apps for social networking & communications, for e-commerce purchases, for travel bookings, to make payments, manage finances, for education, and to entertain ourselves, just to name a few. Likewise, business-focused applications are used across practically all functions and processes in an enterprise.

Today's web applications are mind-bogglingly complex distributed systems. Users of these applications interact through web or mobile front-end user interfaces. But the users rarely see the complex environment consisting of *backend services and software infrastructure components* that respond to user requests made through sleek app user interfaces. Popular consumer apps have thousands of backend services and servers distributed in data centers across the globe. Each feature of an app may be executed on a different server, implemented with a different design choice, written in a different programming language and located in a different geographical location. The seamless in-app user experience makes things look so easy. But developing modern web applications *is anything but easy*.

We use web applications everytime we tweet, watch a movie on Netflix, listen to a song on Spotify, make a travel booking, order food, play an online game, hail a cab, or use any of the numerous online services as part of our daily lives.

Web sites provide information about your business. Web applications

provide services to your customers.

-- Author

In short, without distributed web applications, businesses and modern digital society would come to a grinding halt.

In this book, you will learn the concepts, techniques and tools to design and develop distributed web services and applications using Rust, that communicate over standard internet protocols. Along the way, you will see core Rust concepts in action through practical working examples.

This book is for you if you are a web backend software engineer, full stack application developer, cloud or enterprise architect, CTO for a tech product or simply a curious learner *who is interested in building distributed web applications that are incredibly safe, efficient, highly performant, and do not incur exorbitant costs to operate and maintain*. Through a working example that is progressively built out through the rest of this book, I will show you how to build web services, traditional web application backends in pure Rust.

In this chapter we will review the key characteristics of distributed web applications, understand how and where Rust shines, and outline the example application we will build together in this book.

1.1 Introduction to modern web applications

In this section, we will learn more about the structure of modern, distributed web applications.

Distributed systems have components that may be distributed across several different computing processors, communicate over a network, and concurrently execute workloads. Technically, your home computer itself resembles a networked distributed system (given the modern multi-CPU and multi-core processors).

Popular types of distributed systems include:

1. Distributed networks such as telecommunication networks and the

Internet

2. Distributed client-server applications. Most web-based applications fall in this category
3. Distributed P2P applications such as BitTorrent and Tor
4. Real-time control systems such as air traffic and industrial control
5. Distributed server infrastructures such as cloud, grid and other forms of scientific computing

Distributed systems are broadly composed of three parts: distributed applications networking stack and hardware/OS infrastructure.

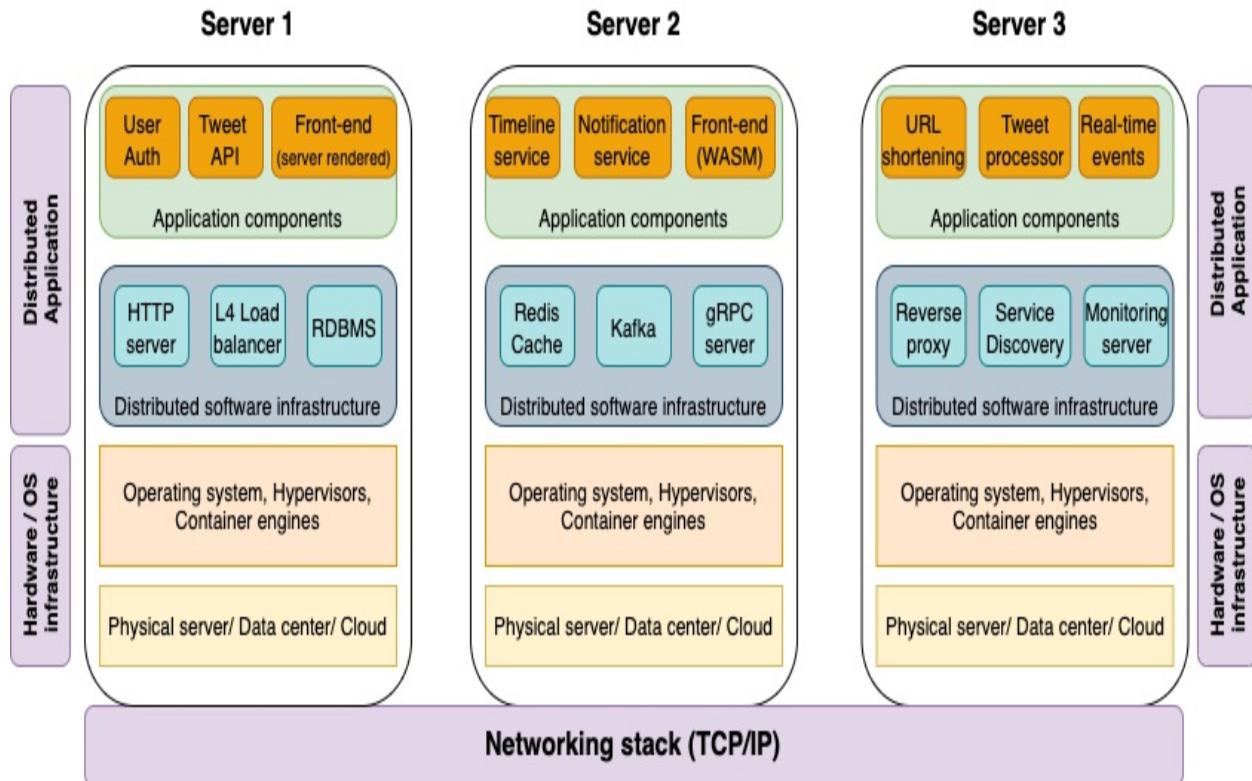
Distributed applications can use a wide array of networking protocols to communicate internally between its components. However, HTTP is the overwhelming choice today for a web service or web application to communicate with the outside world, due to its simplicity and universality.

Web applications are programs that use HTTP as the application-layer protocol, and provide some functionality that is accessible to human users over standard internet browsers. When these web applications are not monolithic, but composed of tens or hundreds of distributed application components that cooperate and communicate over a network, they are called *distributed* web applications. Examples of large-scale distributed web applications include social media applications such as Facebook & Twitter, ecommerce sites such as Amazon or eBay, sharing-economy apps like Uber & Airbnb, entertainment sites such as Netflix, and even user-friendly cloud provisioning applications from providers such as AWS, Google and Azure.

Figure 1.1 provides a representative logical view of the distributed systems stack for a modern web application.

Figure 1.1. Distributed systems stack (simplified)

Example of a distributed systems stack for a social media application



While in the real-world, such systems can be distributed over thousands of servers, in the figure you can see three servers which are connected through a networking stack. These servers may all be within a single data center or distributed on the cloud geographically. Within each server, a layered view of the hardware and software components is shown. A logical breakup of the distributed system is described here:

- **Hardware and OS infrastructure** components such as physical servers (in data center or cloud), operating system, and virtualisation/container runtimes. Devices such as embedded controllers, sensors, and edge devices also can be classified in this layer (think of a futuristic case where tweets are triggered to social media followers of a supermarket chain when stocks of RFID-labelled items are placed or removed from supermarket shelves).
- **Networking stack** comprises the four-layered *Internet Protocol suite* which forms the communication backbone for the distributed system

components to communicate with each other across physical hardware. The four networking layers are (ordered by lowest to highest level of abstraction):

- Network link/access layer,
- Internet layer,
- Transport layer and
- Application layer

The first three layers are implemented at the hardware/OS level on most operating systems. For most distributed web applications, the primary application layer protocol used is HTTP. Popular API protocols such as REST, gRPC and GraphQL use HTTP.

For more details, see the documentation at <https://tools.ietf.org/id/draft-baker-ietf-core-04.html>.

- **Distributed applications:** Distributed applications are a subset of distributed systems. Modern n-tier distributed applications are built as a combination of:
 - *Application front-ends*: these can be mobile apps (running on iOS or Android) or web front-ends running in an internet browser. These app front-ends communicate with application backend services residing on remote servers (usually in a data center or a cloud platform). **End users interact with application front-ends**
 - *Application backends*: These contain the application business rules, database access logic, computation-heavy processes such as image or video processing, and other service integrations. They are deployed as individual processes (such as systemd processes on Unix/Linux) running on physical or virtual machines, or as microservices in container engines (such as Docker) managed by container orchestration environments (such as Kubernetes). Unlike the application front-ends, application backends expose their functionality through application programming interfaces (APIs). **Application front-ends interact with application backend services to complete tasks on behalf of users.**
 - *Distributed software infrastructure* includes components that

provide supporting services for application backends. Examples are protocol servers, databases, KV stores, caching, messaging, load balancers and proxies, service discovery platforms, and other such infrastructure components that are used for communications, operations, security and monitoring of distributed applications.

Application backends interact with distributed software infrastructure for purposes of service discovery, communications, lifecycle support, security and monitoring, to name a few.

Now that we have an overview of distributed web applications, let's take a look at the benefits of using Rust for building them.

1.2 Choosing Rust for web applications

Rust can be used to build all the three layers of distributed applications - *front-ends*, *backend services* and *software infrastructure* components. But each of these layers has a different set of concerns and characteristics to address. It is important to be aware of these while discussing the benefits of Rust.

For example, the client front-ends deal with aspects such as user interface design, user experience, tracking changes in application state and rendering updated views on screen, and constructing and updating DOM.

Considerations while designing backend services include well-designed APIs to reduce roundtrips, high throughput (measured requests per second), response time under varying loads, low and predictable latency for applications such as video streaming and online gaming, low memory and CPU footprint, service discovery and availability.

Software infrastructure layer is concerned primarily with extremely low latencies, low-level control of network and other operating-system resources, frugal usage of CPU and memory, efficient data structures and algorithms, built-in security, small start-up and shut-down time, and ergonomic APIs for usage by application backend services.

As you can see, a single web application comprises components with at least three sets of characteristics and requirements. While each of these is a topic for a separate book in itself, we will look at things more holistically, and focus on a set of common characteristics that broadly benefit all the three layers of a web application.

1.2.1 Characteristics of web applications

Web applications can be of different types.

- Highly mission-critical applications such as autonomous control of vehicles and smart grids, industrial automation, and high-speed trading applications where successful trades depend on ability to quickly and reliably respond to input events
- High-volume transaction and messaging infrastructures such as e-commerce platforms, social networks and retail payment systems
- Near-real time applications such as online gaming servers, video or audio processing, video conferencing and real-time collaboration tools

These applications can be seen to have a common set of requirements which can be expressed as below.

1. Should be safe, secure and reliable
2. Should be resource-efficient
3. Have to minimize latency
4. Should support high concurrency

In addition, the following would be nice-to-have requirements for such services:

1. Should have quick start-up and shut-down time
2. Should be easy to maintain and refactor
3. Must offer developer productivity

It is important to note that all the above requirements can be addressed both at the level of *individual services* and at the *architectural level*. For example, high concurrency can be achieved by an individual service by adopting multi-threading or async I/O as forms of concurrency. Likewise, high concurrency

can be achieved at an architectural level by adding several instances of a service behind a load balancer to process concurrent loads. When we talk of benefits of Rust in this book, we are talking at an *individual service-level*, because architectural-level options are common to all programming languages.

1.2.2 Benefits of Rust for web applications

We've earlier seen that modern web applications comprise web front-ends, backends and software infrastructure. The benefits of Rust for developing web front-ends, either to replace or supplement portions of javascript code, is a hot topic nowadays. It will however not be discussed in this book as this topic deserves a book on its own to be handled appropriately.

Here we will focus primarily on the benefits of Rust for *application backends* and *software infrastructure services*. Rust meets all of the critical requirements that we discussed in the previous section, for such services. Let's see how.

Rust is safe

When we talk about program safety, there are three distinct aspects to consider - *type safety*, *thread safety* and *memory safety*.

Type safety: Rust is a statically typed language. Type checking, which verifies and enforces type constraints, happens at compile-time. The type of a variable has to be known at compile time. If you do not specify a type for a variable, the compiler will try to infer it. If it is unable to do so, or if it sees conflicts, it will let you know and prevent you from proceeding ahead. In this context, Rust is in a similar league as Java, Scala, C and C++. Type safety in Rust is very strongly enforced by the compiler, but with helpful error messages. This helps to safely eliminate an entire class of run-time errors.

Memory safety: Memory safety is, arguably, one of the most unique aspects of the Rust programming language. To do justice to this topic, let's analyze this in detail.

Mainstream programming languages can be classified into two groups based on how they provide memory management.

The first group comprises languages with manual memory management such as C and C++. The second group contains languages with a garbage collector such as Java, C#, Python, Ruby and Go.

Since developers are not perfect, manual memory management also means acceptance of a degree of unsafety, and thus lack of program correctness. So, for cases where low-level control of memory is not necessary and absolute performance is not a must, garbage collection as a technique has become the mainstream feature of many modern programming languages over the last 20 to 25 years. Even though garbage collection has made programs safer than manually managing memory, they come with their limitations in terms of execution speed, consuming additional compute resources, and possible stalling of program execution. Also garbage collection only deals with memory and not other resources such as network sockets, and database handles.

Rust is the first popular language to propose an alternative — automatic memory management and memory safety without garbage collection. As you are probably aware, it achieves this through a unique **ownership model**. Rust enables developers to control the memory layout of their data structures and makes ownership explicit. Rust's ownership model of resource management is modeled around RAII (Resource Acquisition is Initialization)- a C++ programming concept, and smart pointers that enable safe memory usage.

By way of a quick refresher, in this model, each value declared in a Rust program is assigned an owner. Once a value is given away to another owner, it can no longer be used by the original owner. The value is automatically destroyed (memory is deallocated) when the owner of the value goes out of scope.

Rust can also grant temporary access to a value, to another variable or function. This is called *borrowing*. Rust compiler (specifically, the borrow checker) ensures that a *reference to a value* does not outlive the *value being borrowed*. To borrow a value, the & operator is used (called a *reference*). *References* are of two types - *immutable reference* &T, which allows sharing

but not mutation, and *mutable reference* `&mut T`, which allows mutation but not sharing. Rust ensures that whenever there is a mutable borrow of an object, there are no other borrows of that object (either mutable or immutable). All this is enforced at compile time, leading to elimination of entire classes of errors involving invalid memory access.

To summarize, you can program in Rust without fear of invalid memory access, in a language without a garbage collector. Rust provides compile-time guarantees to protect from the following categories of memory safety errors, by default:

1. Null pointer dereferences: Case of a program crashing because a pointer being dereferenced is null
2. Segmentation faults where programs attempt to access a restricted area of memory
3. Dangling pointers, where a value associated with a pointer no longer exists.
4. Buffer overflows, due to programs accessing elements before the start or beyond the end of an array. Rust iterators don't run out of bounds.

Thread safety: In Rust, memory and thread safety (which seem like two completely different concerns) are solved using the same foundational principle of *ownership*. For type safety, Rust ensures no undefined behaviour due to data races, by default. While some of the web development languages may offer similar guarantees, Rust goes one step further and prevents you from sharing objects between threads that are not thread-safe. Rust marks some data types as *thread-safe*, and enforces these for you. Most other languages do not make this distinction between *thread-safe* and *thread-unsafe* data structures. The Rust compiler categorically prevents all types of data races, which makes multi-threaded programs much safer.

Here are a couple of references for a deep-dive into this topic:

- Send and Sync traits: <https://doc.rust-lang.org/book/ch16-04-extensible-concurrency-sync-and-send.html>
- Fearless concurrency with Rust: <https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html>

In addition to what was discussed, there are a few other features of Rust that improve safety of programs:

- All variables in Rust are immutable by default, and explicit declaration is required before mutating any variable. This forces the developer to think through how and where data gets modified, and what is the lifetime of each object.
- Rust's ownership model handles not just memory management, but management of variables owning other resources such as network sockets, database and file handles, and device descriptors.
- Lack of a garbage collector prevents non-deterministic behaviour.
- *Match* clauses (which are equivalent to *Switch* statements in other languages) are exhaustive, which means that the compiler forces the developer to handle every possible variant in the *match* statement, thus preventing developers from inadvertently missing out handling of certain code flow paths that may result in unexpected run-time behaviour.
- Presence of Algebraic data types that make it easier to represent the data model in a concise verifiable manner.

Rust's statically-typed system, ownership & borrowing model, lack of a garbage collector, immutable-by-default values, and exhaustive pattern matching all of which are enforced by the compiler, provide Rust with an undeniable edge for developing safe applications.

Rust is resource-efficient

System resources such as CPU, memory and disk space have progressively become cheaper over the years. While this has proved to be very beneficial in the development and scaling of distributed applications, it also brings a few drawbacks. First of all, there is a general tendency among software teams to simply throw more hardware to solve scalability challenges - more CPU, more memory and more disk space. This is achieved either by adding more CPU/memory/disk resources to the server (vertical scaling, a.k.a *scaling up*) or by adding more machines to the network to share the load (horizontal scaling, a.k.a *scaling out*). But one of the reasons why these have become popular is due to limitations in language design of mainstream web

development languages of today. High level web-development languages such as Javascript, Java, C#, Python and Ruby do not allow fine-grained memory control to limit memory usage. Many programming languages do not utilize multi-core architectures of modern CPUs well. Dynamic scripting languages do not make efficient memory allocations because the type of the variable is known only at run-time, so optimizations are not possible unlike statically-typed languages.

Rust offers the following innate features that enable creation of resource-efficient services:

- Due to its ownership model of memory management, Rust makes it hard (if not impossible) to write code that leaks memory or other resources.
- Rust allows developers to tightly control memory layout for their programs.
- Rust does not have a garbage collector (GC), like a few other mainstream languages, that consumes additional CPU and memory resources. For example, GC code runs in separate threads and consumes resources.
- Rust does not have a large complex runtime. This gives tremendous flexibility to run Rust programs even in underpowered embedded systems and microcontrollers like home appliances and industrial machines. Rust can run in bare metal without kernels.
- Rust discourages deep copy of heap-allocated memory and provides various types of smart pointers to optimize memory footprint of programs. The lack of a runtime in Rust makes it one of the few modern programming languages appropriate for extremely low-resource environments.

Rust combines the best of static typing, fine-grained memory control, efficient use of multi-core CPUs and built-in asynchronous I/O semantics that make it very resource efficient in terms of CPU and memory utilization. All these aspects translate to *lower server costs* and a *lower operational burden* for small and large applications alike.

Rust has low latency

Latency for a roundtrip network request and response depends both on *network latency* and *service latency*. *Network latency* is impacted by many factors such as transmission medium, propagation distance, router efficiency and network bandwidth. *Service latency* is dependent on many factors such as I/O delays in processing the request, whether there is a garbage collector that introduces non-deterministic delays, Hypervisor pauses, amount of context switching (eg in multi-threading), serialization and deserialization costs, etc.

From a purely programming language perspective, Rust provides low latency due to low-level hardware control as a systems programming language. Rust also does not have a garbage collector and run-time, has native support for non-blocking I/O, a good ecosystem of high-performance async (non-blocking) I/O libraries and runtimes, and zero-cost abstractions as a fundamental design principle of the language. Additionally, by default, Rust variables live on the stack which is faster to manage.

Several different benchmarks have shown comparable performance between idiomatic Rust and idiomatic C++ for similar workloads, which is faster than those that can be obtained with mainstream web development languages.

Rust enables fearless concurrency

We previously looked at concurrency features of Rust from a program safety perspective. Now let's look at Rust concurrency from the point of view of better multi-core CPU utilization, throughput and performance for application and infrastructure services.

Rust is a concurrency-friendly language that enables developers to leverage the power of multi-core processors.

Rust provides two types of concurrency - classic multi-threading and asynchronous I/O.

Multi-threading: Rust's traditional multi-threading support provides for both shared-memory and message-passing concurrency. Type-level guarantees are provided for sharing of values. Threads can borrow values, assume ownership and transition the scope of a value to a new thread. Rust also

provides data race safety which prevents thread blocking, improving performance. In order to improve memory efficiency and avoid copying of data shared across threads, Rust provides *reference counting* as a mechanism to track the use of a variable by other processes/threads. The value is dropped when the count reaches zero, which provides for safe memory management. Additionally, *mutexes* are available in Rust for data synchronisation across threads. References to immutable data need not use *mutex*.

Async I/O: Async event-loop based non-blocking I/O concurrency primitives are built into the Rust language with *zero-cost futures* and *async-await*. Non-blocking I/O ensures that code does not hang while waiting for data to be processed.

Further, Rust's rules of immutability provide for high levels of data concurrency.

Rust is a productive language

Even though Rust is first a systems-oriented programming language, it also adds the quality-of-life features of higher-level and functional programming languages.

Here is a (non-exhaustive) list of a few higher-level abstractions in Rust that make for a productive and delightful developer experience:

1. *Closures with anonymous functions.* These capture the environment and can be executed elsewhere (in a different method or thread context). Anonymous functions can be stored inside a variable and can be passed as parameters for functions and across threads.
2. *Iterators*
3. *Generics* and *macros* that provide for code generation and reuse
4. *Enums* such as *Option* and *Result* that are used to express success/failure
5. Polymorphism through *traits*
6. *Dynamic dispatch* through *trait objects*

Rust allows developers to build not just efficient, safe and performant software, but also optimizes for developer productivity with its

expressiveness. It is not without reason that Rust has won the most loved Programming language in the StackOverflow developer survey for five consecutive years: 2016- 2020. The survey can be accessed at: <https://insights.stackoverflow.com/survey/2020>. For more insights into why senior developers love Rust, read this link: <https://stackoverflow.blog/2020/06/05/why-the-developers-who-use-rust-love-it-so-much/>.

We have so far seen how Rust offers a unique combination of memory safety, resource-efficiency, low latency, high concurrency and developer productivity. These impart Rust with the characteristics of *low-level control and speed of a system programming language*, the *developer productivity of higher-level languages* and a very *unique memory model without a garbage collector*. Application backends and infrastructure services directly benefit from these characteristics in order to provide low-latency responses under high loads, while being highly efficient in usage of system resources such as multi-core CPUs and memory. In the next subsection, we will take a look at some of the limitations of Rust.

What does Rust not have?

When it comes to choice of programming languages, there is no *one-size-fits-all*, and no language can be claimed to be suitable for all use cases. Further, due to the nature of programming language design, what may be easy to do in one language could be difficult in another. However, in the interest of providing a complete view to enable decision on using Rust for the web, here are a few things one needs to be cognizant of:

1. *Rust has a steep learning curve.* It is definitely a bigger leap for people who are newcomers to programming, or are coming from dynamic programming or scripting languages. The syntax can be difficult to read at times, even for experienced developers.
2. There are some things that are harder to program in Rust compared to other languages - for example, single and double linked lists. This is due to the way the language is designed.
3. Rust compiler is slower than many other compiled languages, as of this writing. But compilation speed has progressively improved over the last

few years, and work is underway to continually improve this.

4. Rust's ecosystem of libraries and community is still maturing, compared to other mainstream languages.
5. Rust developers are relatively harder to find and hire at scale.
6. Adoption of Rust in large companies and enterprises is still in early days. It does not yet have a natural home to nurture it such as *Oracle* for Java, *Google* for Golang and *Microsoft* for C#.

In this section, we have seen the benefits and drawbacks of using Rust to develop application backend services. In the next section, we will see a preview of the example application that we'll build in this book.

1.3 Visualizing the example application

In this book, we will build web servers, web services and web applications in Rust, and demonstrate concepts through a full-length example. Note that our goal is not to develop a *feature-complete* or *architecture-complete distributed application*, but to learn how to use Rust for the web domain.

We will preview the example application in the next subsection.

1.3.1 What will be build?

EzyTutors - A digital storefront for tutors

Are you a tutor with a unique skill or knowledge that you'd like to monetize? Do you have the necessary time and resources to set up and manage your own website?

EzyTutors is just for you. Take your training business online in just a few minutes.

We will build a digital storefront for tutors to publish their course catalogs online. Tutors can be individuals or training businesses. The digital storefront will be a sales tool for tutors, not a marketplace.

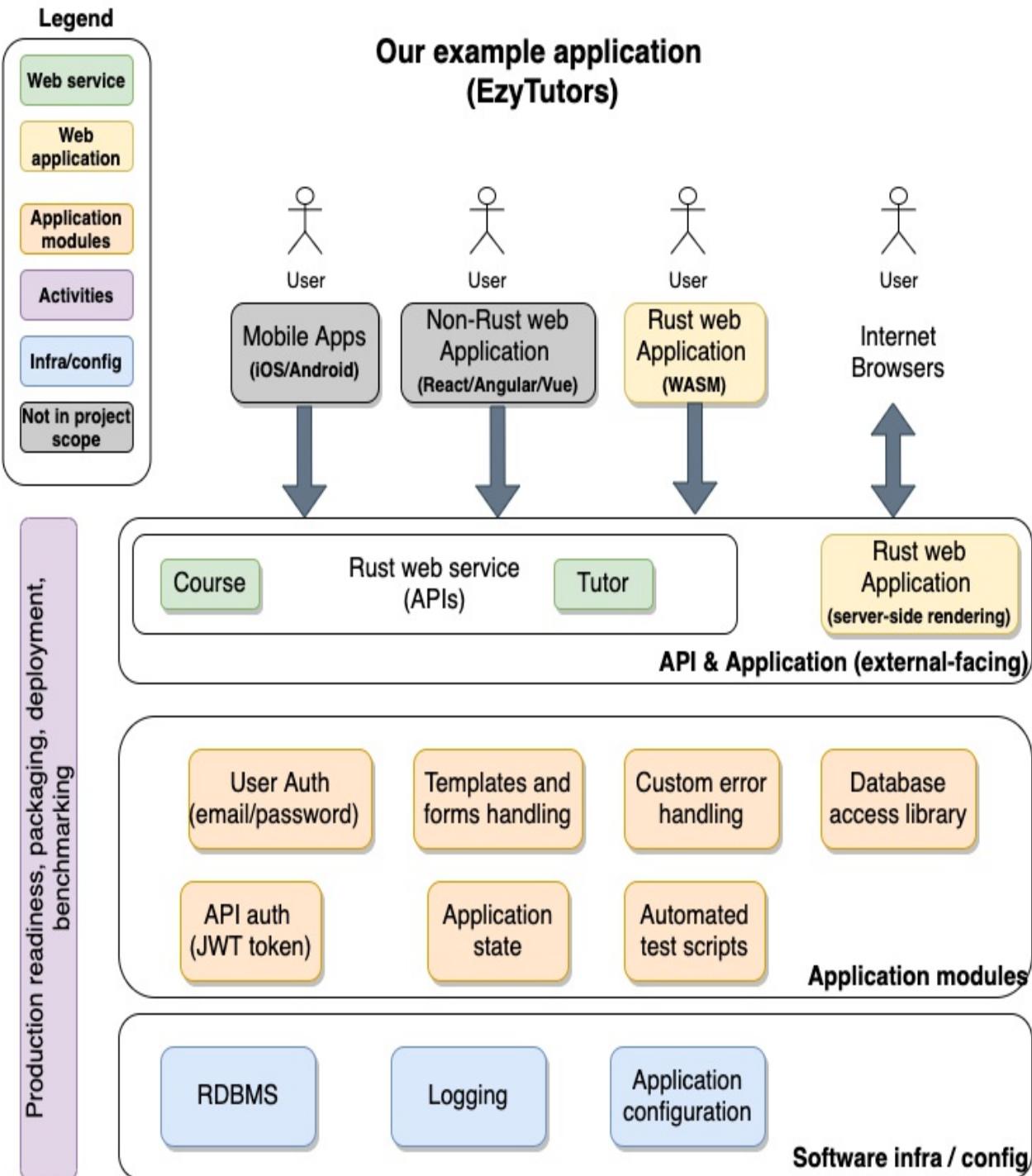
We've defined the product vision. Let's now talk about the scope, followed

by the technical stack.

The storefront will allow tutors to register themselves and then sign in. They can create a course offering and associate it with a course category. A web page with their course list will be generated for each tutor, which they can then share on social media with their network. There will also be a public website that will allow learners to search for courses, browse through courses by tutor, and view course details.

Figure 1.2 shows the logical design of our example application.

Figure 1.2. Our example application



Our technical stack will consist of a web service , a server-rendered web app and a client-rendered web app, all written in *pure Rust*. The course data will be persisted in a relational database. The tools used in this book are *Actix web* for the web framework, *SQLx* for database connections and *Postgres* for the database. Importantly, the design will be asynchronous all the way. Both

Actix web and *SQLx* support full asynchronous I/O, which is very suited for our web application workload that is more I/O heavy than computation-heavy.

We'll first build a web service exposing RESTful APIs that connects to a database, and deals with errors and failures in an application-specific manner. We'll then simulate application lifecycle changes by enhancing the data model, and adding additional functionality, which will require refactoring of code and database migration. This exercise will demonstrate one of the key strengths of Rust, i.e. the ability to fearlessly refactor the code (and reduce technical debt) with the aid of a strongly-typed system and a strict but helpful compiler that has our back.

In addition to the web service, our example will demonstrate how to build a front-end in Rust; the chosen example will be a *server-rendered client app*. We'll use a template engine to render templates and forms for the *server-rendered web application*. It would be possible as well to implement a *WASM-based in-browser app* but such an undertaking falls out of the scope of this book.

Our web application can be developed and deployed on any platform that Rust supports - Linux, Windows and Mac OS. What this means is that we will not use any external library that restricts usage to any specific computing platform. Our application will be capable of being deployed either in a traditional server-based deployment, or in any cloud platform, either as a traditional binary, or in a containerized environment (such as docker and kubernetes).

The chosen problem domain for the example application is a practical scenario, but is not complex to understand. This allows us to focus on the core topic of the book, i.e., *how to apply Rust to the web domain*. As a bonus, we'll also strengthen understanding of Rust by seeing in action concepts such as traits, lifetimes, Result and Option, structs and enums, collections, smart pointers, derivable traits, associated functions and methods, modules and workspaces, unit testing, closures, and functional programming.

This book is about learning the foundations of web development in Rust. What is not covered in this book are topics around how to configure and

deploy additional infrastructural components and tools such as reverse proxy servers, load balancers, firewalls, TLS/SSL, monitoring servers, caching servers , Devops tools, CDNs etc, as these are not Rust-specific topics (but needed for large-scale production deployments).

In addition to building business functionality in Rust, our example application will demonstrate good development practices such as automated tests, code structuring for maintainability, separating configuration from code, generating documentation, and of course, writing idiomatic Rust.

Are you ready for some practical Rust on the web?

1.3.2 Technical guidelines for the example application

This isn't a book about system architecture or software engineering theory. However, I would like to enumerate a few foundational guidelines adopted in the book that will help you better understand the rationale for the design choices made for the code examples in this book.

1. **Project structure:** We'll make heavy use of the Rust module system to separate various pieces of functionality, and keep things organized. We'll use Cargo workspaces to group related projects together, which can include both binaries and libraries.
2. **Single Responsibility principle:** Each logically-separate piece of application functionality should be in its own module. For example, the handlers in the web tier should only deal with processing HTTP messages. The business and database access logic should be in separate modules.
3. **Maintainability:**
 - Variable and function names must be self-explanatory.
 - Keep formatting of code uniform using Rustfmt
 - Write automated test cases to detect and prevent regressions, as the code evolves iteratively.
 - Project structure and file names must be intuitive to understand.
4. **Security:** In this book, we'll cover API authentication using JWT, and password-based user authentication. Infrastructure and network-level

security are not covered. However, it is important to recall that Rust inherently offers memory safety without a garbage collector, and thread-safety that prevents race conditions, thus preventing several classes of hard-to-find and hard-to-fix memory, concurrency and security bugs.

5. **Application Configuration:** Separating configuration from the application is a principle adopted for the example project.
6. **Usage of external crates:** Keep usage of external crates to a minimum. For example, custom error handling functionality is built from scratch in this book, rather than use external crates that simplify and automate error handling. This is because taking short-cuts using external libraries sometimes impedes the learning process and deep understanding.
7. **Async I/O:** It is a deliberate choice to use libraries that support fully asynchronous I/O in the example application, both for network communications and for database access.

Now that we've covered the topics we'll be discussing in the book, the goals of the example project, and the guidelines we'll use to steer design choices, we can start digging into web servers and web services: the topic of our next chapter.

1.4 Summary

- Modern web applications are an indispensable component of digital lives and businesses. But they are complex to build, deploy and operate.
- Distributed web applications comprise *application front-ends*, *backend services* and *distributed software infrastructure*.
- *Application backends* and *software infrastructure* are composed of loosely coupled, cooperative network-oriented services. These have specific run-time characteristics to be satisfied, which have an impact on the tools and technologies used to build them.
- Rust is a highly suitable language to develop distributed web applications, due to its safety, concurrency, low latency and low hardware-resource footprint.
- This book is suitable for readers who are considering Rust for distributed web application development.
- We overviewed the example application we will be building in this book. We also reviewed the key technical guidelines adopted for the

code examples in the book.

2 Writing a basic web server from scratch

This chapter covers

- Writing a TCP server in Rust
- Writing an HTTP server in Rust

In this chapter, you will delve deep into TCP and HTTP communications using Rust.

These protocols are generally abstracted away for developers through higher-level libraries and frameworks used to build web applications. So, why is it important to discuss low level protocols? This would be a fair question.

Learning to work with TCP and HTTP is important because they form the foundation for most communications on the Internet. Popular application communication protocols and techniques such as REST, gRPC, and websockets use HTTP and TCP for transport. Designing and building basic TCP and HTTP servers in Rust gives the confidence to design, develop and troubleshoot higher-level application backend services.

However, if you are eager to get started with the example application, you can move to Chapter 3, and later come back to this chapter at a time appropriate for you.

In this chapter, you will learn the following:

- Write a TCP client and server.
- Build a library to convert between TCP raw byte streams and HTTP messages.
- Build an HTTP server that can serve static web pages (aka *web server*) as well as json data (aka *web service*). Test the server with standard HTTP clients such as cURL (command line) tool and web browser.

Through this exercise, you will understand how Rust data types and traits can be used to model a real-world network protocol, and strengthen your fundamentals of Rust.

The chapter is structured into two sections. In the first section, you will develop a basic network server in Rust that can communicate over TCP/IP. In the second section, you will build a web server that responds to GET requests for web pages and json data. You will achieve all this using just the Rust standard library (no external crates). The HTTP server that you are going to build is not intended to be full-featured or production-ready. But it will serve our stated purpose.

Let's get started.

We spoke about modern applications being constructed as a set of independent components and services, some belonging to the front-end, some backend and some part of the distributed software infrastructure.

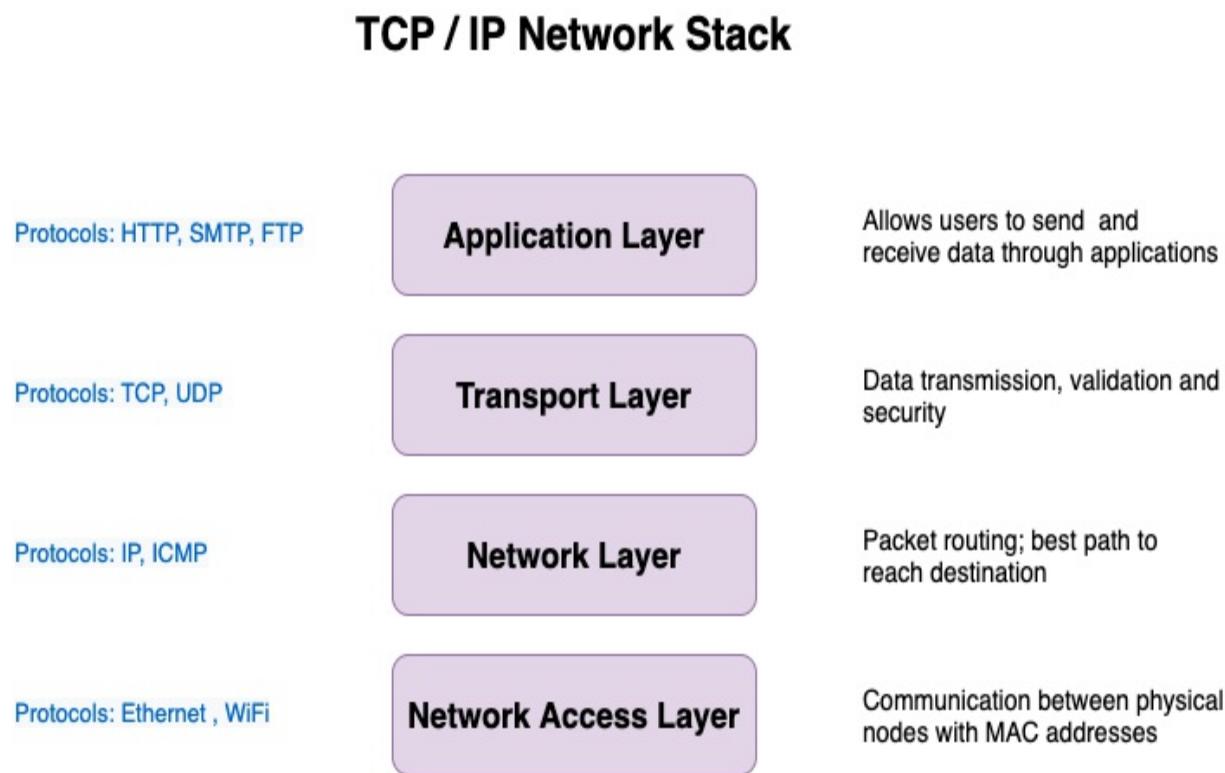
Whenever we have separate components, the question arises as to how these components talk to each other. How does the client (web browser or mobile app) talk to the backend service? How do the backend services talk to the software infrastructure such as databases? This is where the *networking* model comes in.

A *networking* model describes how communication takes place between the sender of a message and its receiver. It addresses questions such as , in what format the message should be sent and received, how the message should be broken up into bytes for physical data transmission, how errors should be handled if data packets do not arrive at the destination etc. The *OSI* model is the most popular networking model, and is defined in terms of a comprehensive seven-layered framework. But for purposes of internet communications, a simplified four-layer model called the *TCP/IP model* is more often adequate to describe how communications take place over the internet between the client making a request and the server that processes that request. The TCP/IP model is described here (<https://www.w3.org/People/Frystyk/thesis/TcpIp.html>).

The *TCP/IP model* is a simplified set of standards and protocols for

communications over the internet. It is organized into four abstract layers: Network Access layer, Internet Layer, Transport Layer and the Application layer, with flexibility on wire protocols that can be used in each layer. The model is named after the two main protocols it is built on- Transmission Control Protocol (TCP) and Internet Protocol(IP). This is shown in figure 2.1. The main thing to note is that these four layers complement each other in ensuring that a message is sent successfully from the sending process to the receiving process.

Figure 2.1. TCP/IP network model



We will now look at the role of each of these four layers in communications.

The *Application layer* is the highest layer of abstraction. The semantics of the message are understood by this layer. For example, a web browser and web server communicate using HTTP, or an email client and email server communicate using SMTP(Simple Mail Transfer Protocol). There are other such protocols such as DNS (Domain Name Service) and FTP (File Transfer Protocol). All these are called application-layer protocols because they deal with specific user applications - such as web browsing, emails or file

transfers. *In this book, we will focus mainly on the HTTP protocol at the application layer.*

The *Transport layer* provides reliable end-to-end communication. While the application layer deals with messages that have specific semantics (such as sending a GET request to get shipment details), the transport protocols deal with sending and receiving raw bytes. (Note: all application layer protocol messages eventually get converted into raw bytes for transmission by the transport layer). TCP and UDP are the two main protocols used in this layer, with QUIC (Quick UDP Internet Connection) also being a recent entrant. TCP is a connection-oriented protocol that allows data to be partitioned for transmission and reassembled in a reliable manner at the receiving end. UDP is a connectionless protocol and does not provide guarantees on delivery, unlike TCP. UDP is consequently faster and suitable for certain class of applications eg DNS lookups, voice or video applications. *In this book, we will focus on the TCP protocol for transport layer.*

The *Network layer* uses IP addresses and routers to locate and route packets of information to hosts across networks. While the TCP layer is focused on sending and receiving raw bytes between two servers identified by their IP addresses and port numbers, the network layer worries about what is the best path to send data packets from source to destination. *We do not need to directly work with the network layer as Rust's standard library provides the interface to work with TCP and sockets, and handles the internals of network layer communications.*

The *Network Access layer* is the lowest layer of the TCP/IP network model. It is responsible for transmission of data through a physical link between hosts, such as by using network cards. *For our purposes, it does not matter what physical medium is used for network communications.*

Now that we have an overview of the TCP/IP networking model, we'll learn how to use the TCP/IP protocol to send and receive messages in Rust.

2.1 Writing a TCP server in Rust

In this section, you will learn how to perform basic TCP/IP networking

communications in Rust, fairly easily. Let's start by understanding how to use the TCP/IP constructs in the Rust standard library.

2.1.1 Designing the TCP/IP communication flow

The Rust standard library provides networking primitives through the **std::net** module for which documentation can be found at: <https://doc.rust-lang.org/std/net/>. This module supports basic TCP and UDP communications. There are two specific data structures, **TcpListener** and **TcpStream**, which have the bulk of the methods needed to implement our scenario.

Let us see how to use these two data structures.

TcpListener is used to create a TCP socket server that binds to a specific port. A client can send a message to a socket server at the specified socket address (combination of IP address of the machine and port number). There may be multiple TCP socket servers running on a machine. When there is an incoming network connection on the network card, the operating system routes the message to the right TCP socket server using the port number.

Example code to create a socket server is shown here.

```
use std::net::TcpListener;  
  
let listener = TcpListener::bind("127.0.0.1:3000")
```

After binding to a port, the socket server should start to listen for the next incoming connection. This is achieved as shown here:

```
listener.accept()
```

For listening continually (in a loop) for incoming connections, the following method is used:

```
listener.incoming()
```

The *listener.incoming()* method returns an iterator over the connections received on this listener. Each connection represents a stream of bytes of type

TcpStream. Data can be transmitted or received on this *TcpStream* object. Note that reading and writing to *TcpStream* is done in raw bytes. Code snippet is shown next.(Note: error handling is excluded for simplicity)

```
for stream in listener.incoming() {  
    //Read from stream into a bytes buffer  
    stream.read(&mut [0;1024]);  
    // construct a message and write to stream  
    let message = "Hello".as_bytes();  
    stream.write(message)  
}
```

Note that

- for *reading from a stream*, we have constructed a bytes buffer (called *byte slice* in Rust).
- for *writing to a stream*, we have constructed a *string slice* and converted it to a *byte slice* using *as_bytes()* method

So far, we've seen the server side of TCP socket server. On the client side, a connection can be established with the TCP socket server as shown:

```
let stream = TcpStream.connect("172.217.167.142:80")
```

To recap, *connection management* functions are available from the *TcpListener* struct of the *std::net* module. To read and write on a connection, *TcpStream* struct is used.

Let's now apply this knowledge to write a working TCP client and server.

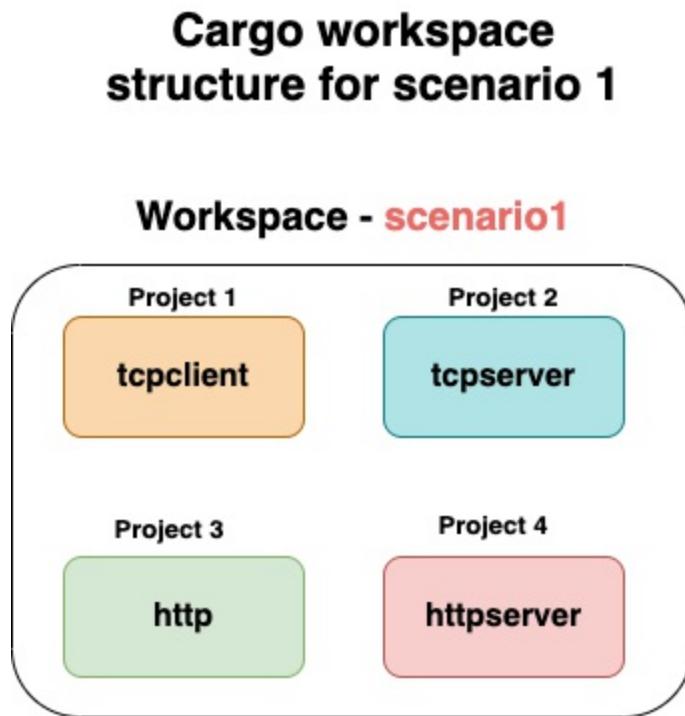
2.1.2 Writing the TCP server and client

Let's first setup a project structure . **Figure 2.2** shows the workspace called *scenario1* which contains four projects - *tcpclient*, *tcpserver*, *http* and *httpserver*.

For Rust projects, a *workspace* is a container project which *holds* other projects. The benefit of the *workspace* structure is that it enables us to manage multiple projects as one unit. It also helps to store all related projects seamlessly within a single git repo. We will create a *workspace* project called

scenario1. Under this workspace, we will create four new Rust projects using *cargo*, the Rust project build and dependencies tool. The four projects are *tcpclient*, *tcpserver*, *http* and *httpserver*.

Figure 2.2. Cargo workspace project



The commands for creating the workspace and associated projects are listed here.

Start a new cargo project with:

```
cargo new scenario1 && cd scenario1
```

The *scenario1* directory can also be referred to as the workspace root.

Under *scenario1* directory, create the following four new Rust projects:

```
cargo new tcpserver
cargo new tcpclient
cargo new httpserver
cargo new --lib http
```

- *tcpserver* will be the *binary* project for TCP server code

- *tcpclient* will be the *binary* project for TCP client code
- *httpserver* will be the *binary* project for HTTP server code
- *http* will be the *library* project for http protocol functionality

Now that the projects are created, we have to declare *scenario1* project as a workspace and specify its relationship with the four subprojects. Add the following:

Listing 2.1. scenario1/Cargo.toml

```
[workspace]
members = [
    "tcpserver", "tcpclient", "http", "httpserver",
]
```

We will now write the code for TCP server and client in two iterations:

1. In the first iteration, we will write the TCP server and client to do a sanity check that connection is being established from client to server.
2. In the second iteration, we will send a text from client to server and have the server echo it back.

General note about following along with the code

Many of the code snippets shown in this chapter (and across the book) have inline numbered code annotations to describe the code. If you are copying and pasting code (from any chapter in this book) into your code editor, ensure to remove the code annotation numbers (or the program will not compile). Also, the pasted code may sometimes be misaligned, so manual verification may be needed to compare pasted code with the code snippets in the chapter, in case of compilation errors.

Iteration 1

Go to *tcpserver* folder and modify *src/main.rs* as follows:

Listing 2.2. First iteration of TCP server (*tcpserver/src/main.rs*)

```

use std::net::TcpListener;

fn main() {
    let connection_listener = TcpListener::bind("127.0.0.1:3000");
    println!("Running on port 3000");
    for stream in connection_listener.incoming() { #2
        let _stream = stream.unwrap(); #
        println!("Connection established");
    }
}

```

From root folder of workspace (*scenario1*), run :

```
cargo run -p tcpserver #1
```

The server will start and the message *Running on port 3000* is printed to the terminal. We now have a working TCP server listening on port 3000 on localhost.

Let's next write a TCP client to establish connection with the TCP server.

Listing 2.3. tcpclient/src/main.rs

```

use std::net::TcpStream;

fn main() {
    let _stream = TcpStream::connect("localhost:3000").unwrap();
}

```

In a new terminal, from root folder of workspace, run :

```
cargo run -p tcpclient
```

You will see the message "connection established" printed to terminal where the TCP server is running as shown:

```
Running on port 3000
Connection established
```

We now have a TCP server running on port 3000, and a TCP client that can establish connection to it.

We now can try sending a message from our client and have the server echo

it back.

Iteration 2:

Modify the `tcpserver/src/main.rs` file as follows:

Listing 2.4. Completing the TCP server

```
use std::io::{Read, Write};      #1
use std::net::TcpListener;
fn main() {
    let connection_listener = TcpListener::bind("127.0.0.1:3000")
    println!("Running on port 3000");
    for stream in connection_listener.incoming() {
        let mut stream = stream.unwrap();    #2
        println!("Connection established");
        let mut buffer = [0; 1024];
        stream.read(&mut buffer).unwrap();   #3
        stream.write(&mut buffer).unwrap(); #4
    }
}
```

In the code shown, we are echoing back to the client, whatever we receive from it. Run the TCP server with **cargo run -p tcpserver** from the workspace root directory.

Read and Write traits

Traits in Rust define shared behaviour. They are similar to *interfaces* in other languages, with some differences. The Rust standard library(`std`) defines several traits that are implemented by data types within `std`. These traits can also be implemented by user-defined data types such as *structs* and *enums*.

Read and *Write* are two such traits defined in the Rust standard library.

Read trait allows for reading bytes from a source. Examples of sources that implement the *Read* trait include *File*, *Stdin* (standard input), and *TcpStream*. Implementers of the *Read* trait are required to implement one method - *read()*. This allows us to use the same *read()* method to read from a *File*, *Stdin*, *TcpStream* or any other type that implements the *Read* trait.

Similarly, the *Write* trait represents objects that are byte-oriented sinks. Implementers of the *Write* trait implement two methods - *write()* and *flush()*. Examples of types that implement the *Write* trait include *File*, *Stderr*, *Stdout* and *TcpStream*. This trait allows us to write to either a *File*, *standard output*, *standard error* or *TcpStream* using the *write()* method.

The next step is to modify the TCP client to send a message to the server, and then print what is received back from the server. Modify the file *tcpclient/src/main.rs* as follows:

Listing 2.5. Completing the TCP client

```
use std::io::{Read, Write};
use std::net::TcpStream;
use std::str;

fn main() {
    let mut stream = TcpStream::connect("localhost:3000").unwrap()
    stream.write("Hello".as_bytes()).unwrap(); #1
    let mut buffer = [0; 5];
    stream.read(&mut buffer).unwrap();           #2
    println!(
        "Got response from server:{:?}",          #3
        str::from_utf8(&buffer).unwrap()
    );
}
```

Run the TCP client with **cargo run -p tcpclient** from the workspace root. Make sure that the TCP Server is also running in another terminal window.

You will see the following message printed to the terminal window of the TCP client:

```
Got response from server:"Hello"
```

Congratulations. You have written a TCP server and a TCP client that can communicate with each other.

Result type and unwrap() method

In Rust, it is idiomatic for a function or method that can fail to return a

Result<T,E> type. This means the *Result* type wraps another data type T in case of success, or wraps an *Error* type in case of failure, which is then returned to the calling function. The calling function in turn inspects the *Result* type and unwraps it to receive either the value of type T or type *Error* for further processing.

In the examples so far, we have made use of the *unwrap()* method in several places, to retrieve the value embedded within the *Result* object by the standard library methods. *unwrap()* method returns the value of type T if operation is successful , or panics in case of error. In a real-world application, this is not the right approach, as *Result* type in Rust is for recoverable failures, while panic is used for unrecoverable failures. However, we have used it because use of *unwrap()* simplifies our code for learning purposes. We will cover proper error handling in later chapters.

In this section, we have learnt how to do TCP communications in Rust. You have also noticed that TCP is a low-level protocol which only deals in byte streams. It does not have any understanding of the semantics of messages and data being exchanged. For writing web applications, semantic messages are easier to deal with than raw byte streams. So, we need to work with a higher-level application protocol such as HTTP, rather than TCP. This is what we will look at in the next section.

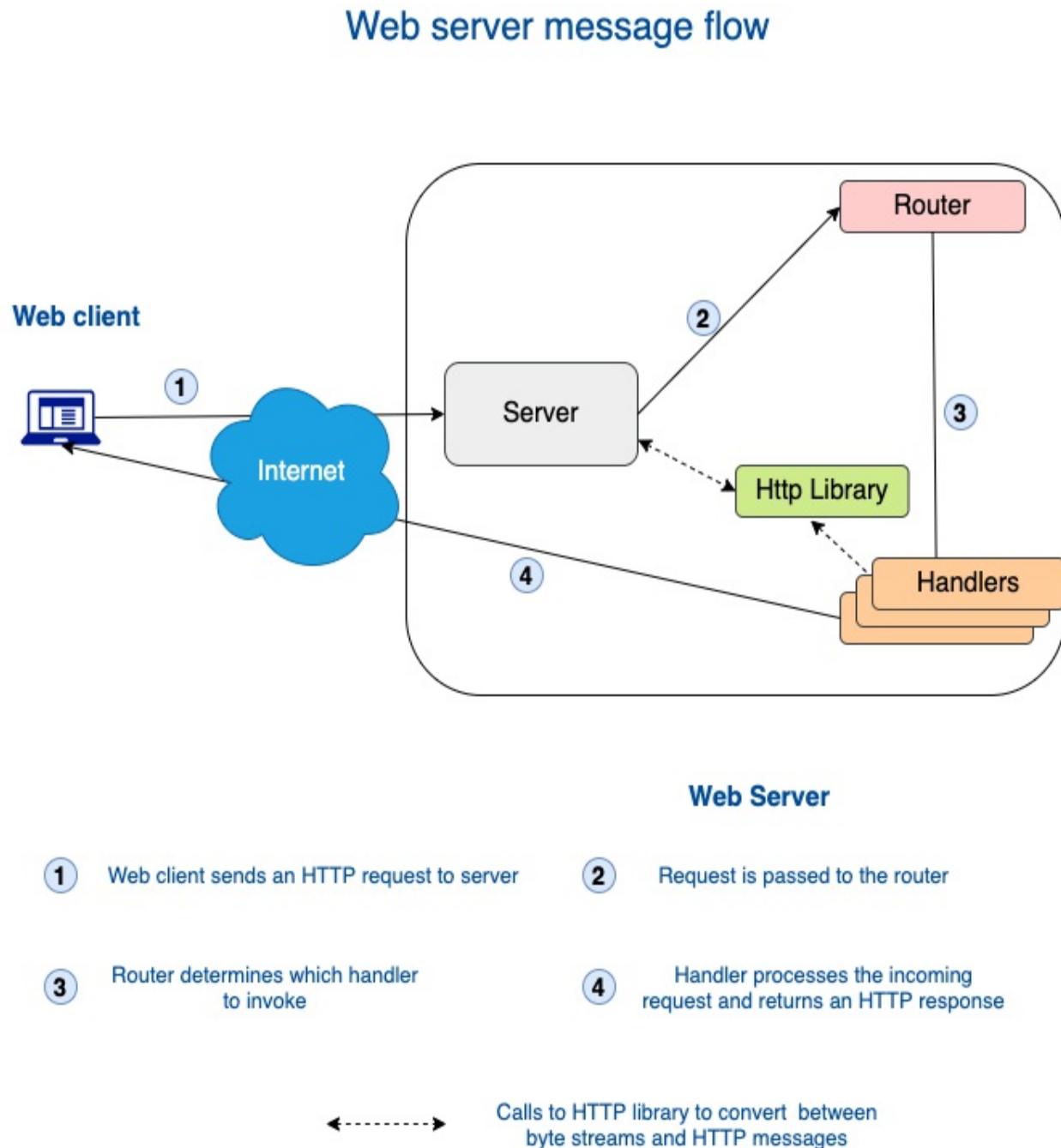
2.2 Writing an HTTP server in Rust

In this section, we'll build a web server in Rust that can communicate with HTTP messages.

But Rust does not have built-in support for HTTP. There is no **std::http** module that we can work with. Even though there are third-party HTTP crates available, we'll write one from scratch. Through this, we will learn how to apply Rust for developing lower-level libraries and servers, that modern web applications in turn rely upon.

Let's first visualize the features of the web server that we are going to build. The communication flow between the client and the various modules of the web server is depicted in figure 2.3.

Figure 2.3. Web server message flow



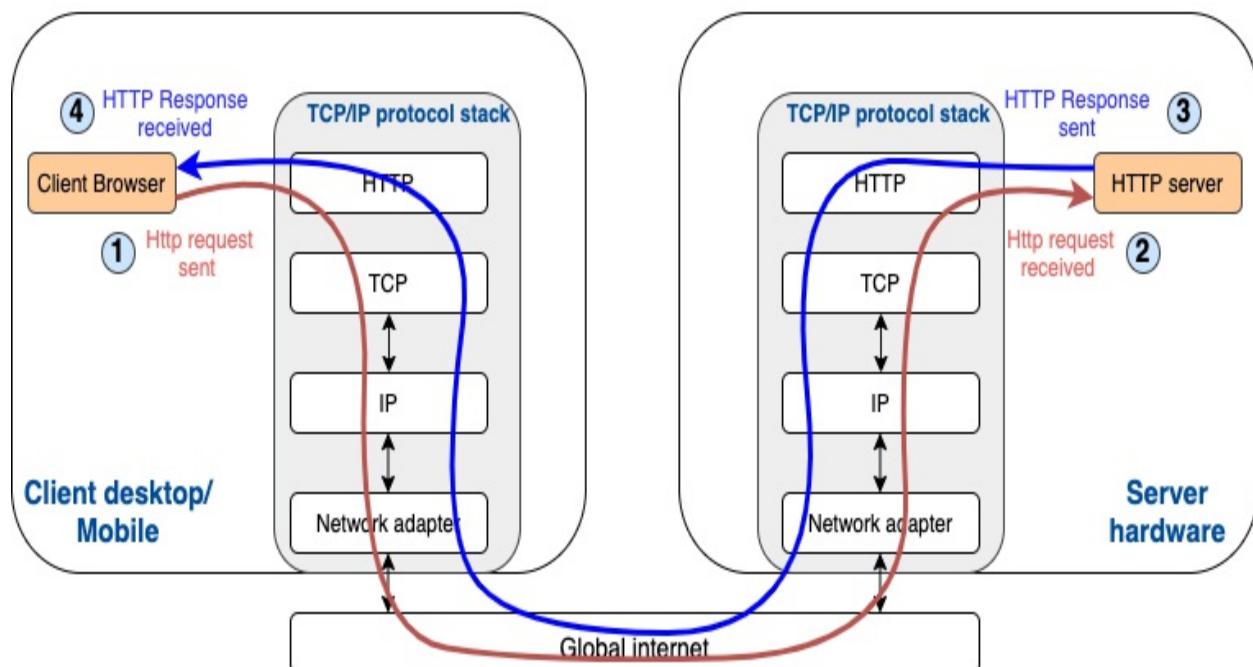
Our Web server will have four components - *Server*, *Router*, *Handler* and *HTTP library*. Each of these components has a specific purpose, in line with *Single Responsibility Principle* (SRP). The **Server** listens for incoming TCP byte streams. The **HTTP library** interprets the byte stream and converts it to *HTTP Request* (message). The **router** accepts an *HTTP Request* and

determines which handler to invoke. The **handler** processes the *HTTP request* and constructs an *HTTP response*. The *HTTP response* message is converted back to a byte stream using the HTTP library, which is then sent back to the client.

Figure 2.4 shows another view of the HTTP client-server communications, this time depicting how the HTTP messages flow through the *TCP/IP protocol stack*. The TCP/IP communications are handled at the operating system level both at the client and server side, and a web application developer only works with HTTP messages.

Figure 2.4. HTTP communications with protocol stack

HTTP communications - protocol stack



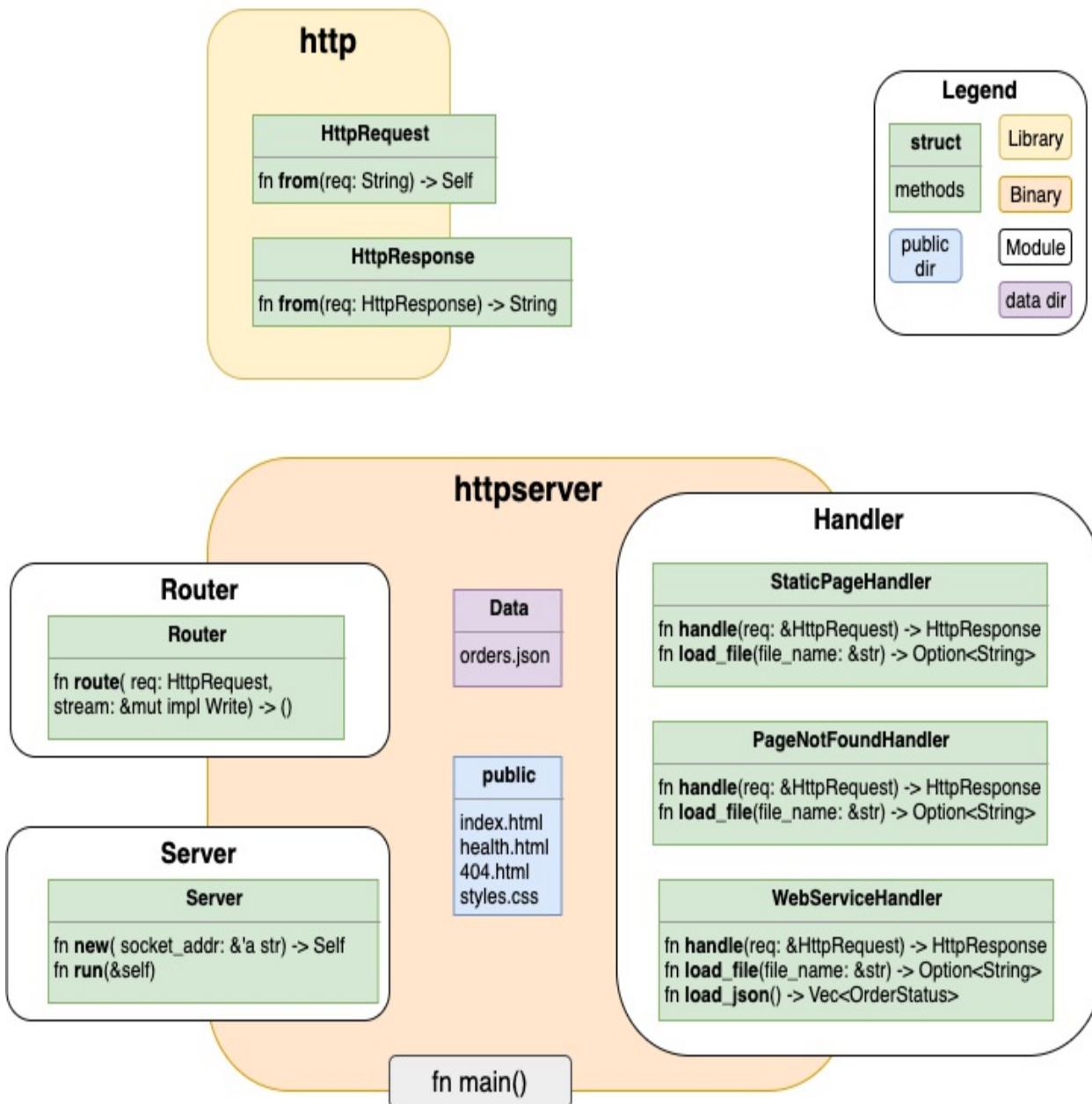
Let's build the code in the following sequence:

- Build the *HTTP library*
- Write the *main()* function for the project
- Write the *server* module
- Write the *router* module
- Write the *handler* module

For convenience, **figure 2.5** shows a summary of the code design, showing the key modules, structs and methods for the *http* library and *httpserver* project.

Figure 2.5. Design overview of web server

Design of Web server



We'll be writing code for the modules, structs and methods shown in this figure. Here is a short summary of what each component in the figure does:

- **http**: Library containing types *HttpRequest* and *HttpResponse*. It

implements the logic for converting between HTTP requests and responses, and corresponding Rust data structures.

- **httpserver**: Main web server that incorporates a main() function, socket server, handler and router, and manages the coordinations among them. It serves as both a web server (serving html) and a web service(serving json).

Shall we get started?

2.2.1 Parsing HTTP request messages

In this section we will build an HTTP library. The library will contain data structures and methods to do the following:

- Interpret an incoming byte stream and convert it into an HTTP Request message
- Construct an HTTP response message and convert it into a byte stream for transmitting over the wire

We are now ready to write some code.

Recall that we have already created a library called *http* under *scenario1* workspace.

The code for HTTP library will be placed under *http/src* folder.

In *http/src/lib.rs*, add the following code:

```
pub mod httprequest;
```

This tells the compiler that we are creating a new publicly-accessible module called *httprequest* in the *http* library.

Also, delete the pre-generated test script (by cargo tool) from this file. We'll write test cases later.

Create two new files *httprequest.rs* and *httpresponse.rs* under *http/src*, to contain the functionality to deal with HTTP requests and responses

respectively.

We will start with designing the Rust data structures to hold an HTTP request. When there is an incoming byte stream over a TCP connection, we will parse it and convert it into strongly-typed Rust data structures for further processing. Our HTTP server program can then work with these Rust data structures, rather than with TCP streams.

Table 1 shows a summary of Rust data structures needed to represent an incoming HTTP request:

Table 2.1. Table showing the list of data structures we will be building.

Data structure name	Rust data type	Description
HttpRequest	struct	Represents an HTTP request
Method	enum	Specifies the allowed values (variants) for HTTP Methods
Version	enum	Specifies allowed values for HTTP Versions

We'll implement a few traits on these data structures, to impart some behaviour. **Table 2** shows a description of the traits we will implement on the three data structures.

Table 2.2. Table showing the list of traits implemented by the data structures for HTTP requests.

--	--

Rust trait implemented	Description
From<&str>	This trait enables conversion of incoming string slice into <i>HttpRequest</i> data structure
Debug	Used to print debug messages
PartialEq	Used to compare values as part of parsing and automated test scripts

Let's now convert this design into code. We'll write the data structures and methods.

Method

We will code the *Method* enum and trait implementations here.

Add the following code to *http/src/httprequest.rs*.

The code for **Method** enum is shown here. We use an enum data structure as we want to allow only predefined values for the HTTP method in our implementation. We will only support two HTTP methods in this version of implementation- **GET** and **POST** requests. We'll also add a third type - **Uninitialized**, to be used during initialization of data structures in the running program.

Add the following code to *http/src/httprequest.rs*:

```
#[derive(Debug, PartialEq)]
pub enum Method {
    Get,
    Post,
    Uninitialized,
}
```

The trait implementation for **Method** is shown here (to be added to `httprequest.rs`):

```
impl From<&str> for Method {
    fn from(s: &str) -> Method {
        match s {
            "GET" => Method::Get,
            "POST" => Method::Post,
            _ => Method::Uninitialized,
        }
    }
}
```

Implementing the **from** method in **From** trait enables us to read the *method* string from the HTTP request line, and convert it into `Method::Get` or `Method::Post` variant. In order to understand the benefit of implementing this trait and to test if this method works, let's write some test code. Add the following to `http/src/httprequest.rs`:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_method_into() {
        let m: Method = "GET".into();
        assert_eq!(m, Method::Get);
    }
}
```

From the workspace root, run the following command:

```
cargo test -p http
```

You will notice a message similar to this stating that the test has passed.

```
running 1 test
test httprequest::tests::test_method_into ... ok

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

The string "GET" is converted into `Method::Get` variant using just the `.into()` syntax, which is the benefit of implementing the *From* trait. It makes for clean, readable code.

Let's now look at the code for the *Version* enum.

Version

The definition of **Version** enum is shown next. We will support two HTTP versions just for illustration though we will be working only with HTTP/1.1 for our examples. There is also a third type - **Uninitialized**, to be used as default initial value.

Add the following code to *http/src/httprequest.rs*:

```
#[derive(Debug, PartialEq)]
pub enum Version {
    V1_1,
    V2_0,
    Uninitialized,
}
```

The trait implementation for **Version** is similar to that for *Method* enum (to be added to *httprequest.rs*).

```
impl From<&str> for Version {
    fn from(s: &str) -> Version {
        match s {
            "HTTP/1.1" => Version::V1_1,
            _ => Version::Uninitialized,
        }
    }
}
```

Implementing the **from** method in **From** trait enables us to read the HTTP protocol version from the incoming HTTP request, and convert it into a *Version* variant.

Let's test if this method works. Add the following to *http/src/httprequest.rs* inside the previously-added **mod tests** block (after the *test_method_into()* function), and run the test from the workspace root with **cargo test -p http** :

```
#[test]
fn test_version_into() {
    let m: Version = "HTTP/1.1".into();
    assert_eq!(m, Version::V1_1);
```

```
}
```

You will see the following message on your terminal:

```
running 2 tests
test httprequest::tests::test_method_into ... ok
test httprequest::tests::test_version_into ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 fil
```

Both the tests pass now. The string "HTTP/1.1" is converted into **Version::V1_1** variant using just the `.into()` syntax, which is the benefit of implementing the *From* trait.

HttpRequest

This represents the complete HTTP request. The structure is shown in code here. Add this code to the beginning of the file `http/src/httprequest.rs`.

Listing 2.6. Structure of HTTP request

```
use std::collections::HashMap;

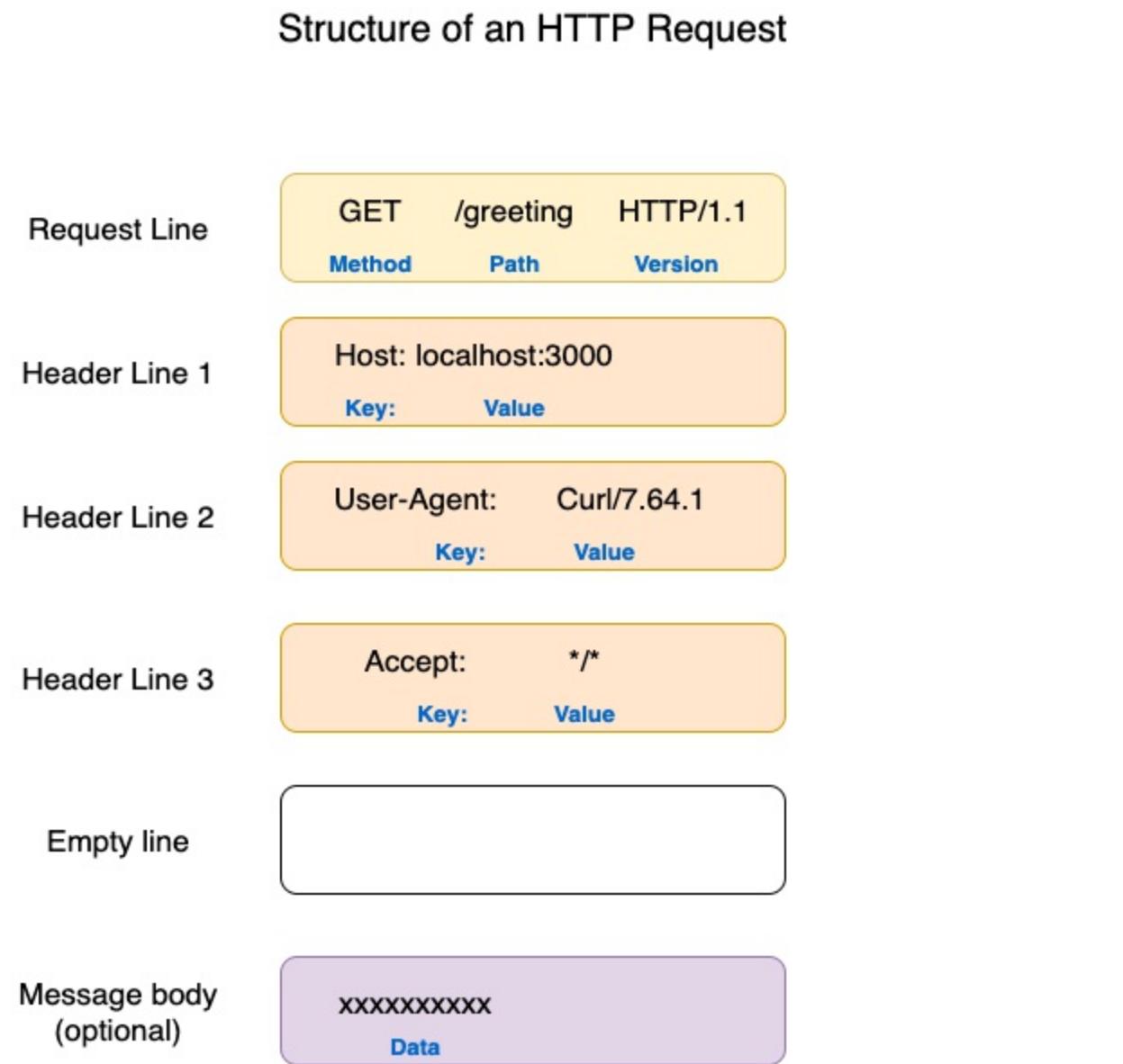
#[derive(Debug, PartialEq)]
pub enum Resource {
    Path(String),
}

#[derive(Debug)]
pub struct HttpRequest {
    pub method: Method,
    pub version: Version,
    pub resource: Resource,
    pub headers: HashMap<String, String>,
    pub msg_body: String,
}
```

The `From<&str>` trait implementation for `HttpRequest` struct is at the core of our exercise. What this enables us to do is to convert the incoming request into a Rust HTTP Request data structure that is convenient to process further.

Figure 2.6 shows the structure of a typical HTTP request.

Figure 2.6. Structure of HTTP request



The figure shows a sample HTTP request consisting of a request line, a set of one or more header lines followed by a blank line, and then an optional message body. We'll have to parse all these lines and convert them into our `HTTPRequest` type. That is going to be the job of the `from()` function as part of the `From<&str>` trait implementation.

The core logic for the `From<&str>` trait implementation is listed here:

1. Read each line in the incoming HTTP request. Each line is delimited by

CRLF (\r\n).

2. Evaluate each line as follows:

- If the line is a request line (we are looking for the keyword *HTTP* to check if it is a request line as all request lines contain *HTTP* keyword and version number), extract the method, path and *HTTP* version from the line.
- If the line is a header line (identified by separator ':'), extract *key* and *value* for the header item and add them to the list of headers for request. Note there can be multiple header lines in an *HTTP* request. To keep things simple, let's make the assumption that the *key* and *value* must be composed of printable ASCII characters (i.e., characters that have values between 33 and 126 in base 10, except colon).
- If a line is empty (\n\r), then treat it as a separator line. No action is needed in this case
- If the message body is present, then scan and store it as String.

Add the following code to *http/src/httprequest.rs*.

Let's look at the code in smaller chunks. First, here is the skeleton of the code. Don't type this in yet, this is just to show the structure of code.

```
impl From<String> for HttpRequest {  
    fn from(req: String) -> Self {}  
}  
fn process_req_line(s: &str) -> (Method, Resource, Version) {}  
fn process_header_line(s: &str) -> (String, String) {}
```

We have a *from()* method that we should implement for the *From* trait. There are two other supporting functions for parsing request line and header lines respectively.

Let's first look at the *from()* method. Add the following to *httprequest.rs*.

Listing 2.7. Parsing incoming HTTP requests: from() method

```
impl From<String> for HttpRequest {  
    fn from(req: String) -> Self {
```

```

let mut parsed_method = Method::Uninitialized;
let mut parsed_version = Version::V1_1;
let mut parsed_resource = Resource::Path("".to_string());
let mut parsed_headers = HashMap::new();
let mut parsed_msg_body = "";

// Read each line in the incoming HTTP request
for line in req.lines() {
    // If the line read is request line, call function pr
    if line.contains("HTTP") {
        let (method, resource, version) = process_req_lin
        parsed_method = method;
        parsed_version = version;
        parsed_resource = resource;
    // If the line read is header line, call function pro
    } else if line.contains(":") {
        let (key, value) = process_header_line(line);
        parsed_headers.insert(key, value);
    // If it is blank line, do nothing
    } else if line.len() == 0 {
        // If none of these, treat it as message body
    } else {
        parsed_msg_body = line;
    }
}
// Parse the incoming HTTP request into HttpRequest struct
HttpRequest {
    method: parsed_method,
    version: parsed_version,
    resource: parsed_resource,
    headers: parsed_headers,
    msg_body: parsed_msg_body.to_string(),
}
}
}

```

Based on the logic described earlier, we are trying to detect the various types of lines in the incoming HTTP Request, and then constructing an *HttpRequest* struct with the parsed values. We'll look at the two supporting methods next.

Here is the code for processing the request line of the incoming request. Add it to *httprequest.rs*, after the *impl From<String> for HttpRequest {}* block.

Listing 2.8. Parsing incoming HTTP requests: `process_req_line()` function

```

fn process_req_line(s: &str) -> (Method, Resource, Version) {
    // Parse the request line into individual chunks split by whitespace
    let mut words = s.split_whitespace();
    // Extract the HTTP method from first part of the request line
    let method = words.next().unwrap();
    // Extract the resource (URI/URL) from second part of the request line
    let resource = words.next().unwrap();
    // Extract the HTTP version from third part of the request line
    let version = words.next().unwrap();

    (
        method.into(),
        Resource::Path(resource.to_string()),
        version.into(),
    )
}

```

And here is the code for parsing the header line. Add it to *httprequest.rs* after *process_req_line()* function.

Listing 2.9. Parsing incoming HTTP requests: *process_header_line()* function

```

fn process_header_line(s: &str) -> (String, String) {
    // Parse the header line into words split by separator ':'
    let mut header_items = s.split(":");
    let mut key = String::from("");
    let mut value = String::from("");
    // Extract the key part of the header
    if let Some(k) = header_items.next() {
        key = k.to_string();
    }
    // Extract the value part of the header
    if let Some(v) = header_items.next() {
        value = v.to_string()
    }

    (key, value)
}

```

This completes the code for the *From* trait implementation for the *HTTPRequest* struct.

Let's write a unit test for the HTTP request parsing logic in *http/src/httprequest.rs*, inside **mod tests** (tests module). Recall that we've

already written the test functions `test_method_into()` and `test_version_into()` in the `tests` module. The `tests` module should look like this at this point in `httprequest.rs` file:

```
#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_method_into() {
        let m: Method = "GET".into();
        assert_eq!(m, Method::Get);
    }

    #[test]
    fn test_version_into() {
        let m: Version = "HTTP/1.1".into();
        assert_eq!(m, Version::V1_1);
    }
}
```

Now add another test function to the same `tests` module in the file, after the `test_version_into()` function.

Listing 2.10. Test scripts for parsing HTTP requests

```
#[test]
fn test_read_http() {
    let s: String = String::from("GET /greeting HTTP/1.1\r\nH
let mut headers_expected = HashMap::new(); #2
headers_expected.insert("Host".into(), " localhost".into(
headers_expected.insert("Accept".into(), " */*".into());
headers_expected.insert("User-Agent".into(), " curl/7.64.
let req: HttpRequest = s.into(); #3
assert_eq!(Method::Get, req.method); #4
assert_eq!(Version::V1_1, req.version); #5
assert_eq!(Resource::Path("/greeting".to_string()), req.r
assert_eq!(headers_expected, req.headers); #6
}
```

Run the test with **cargo test -p http** from the workspace root folder.

You should see the following message indicating that all the three tests have passed:

```
running 3 tests
```

```
test httprequest::tests::test_method_into ... ok
test httprequest::tests::test_version_into ... ok
test httprequest::tests::test_read_http ... ok

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

We have completed the code for HTTP request processing. This library is able to parse an incoming HTTP GET or POST message, and convert it into a Rust data struct.

Let's now write the code to process HTTP responses.

2.2.2 Constructing HTTP response messages

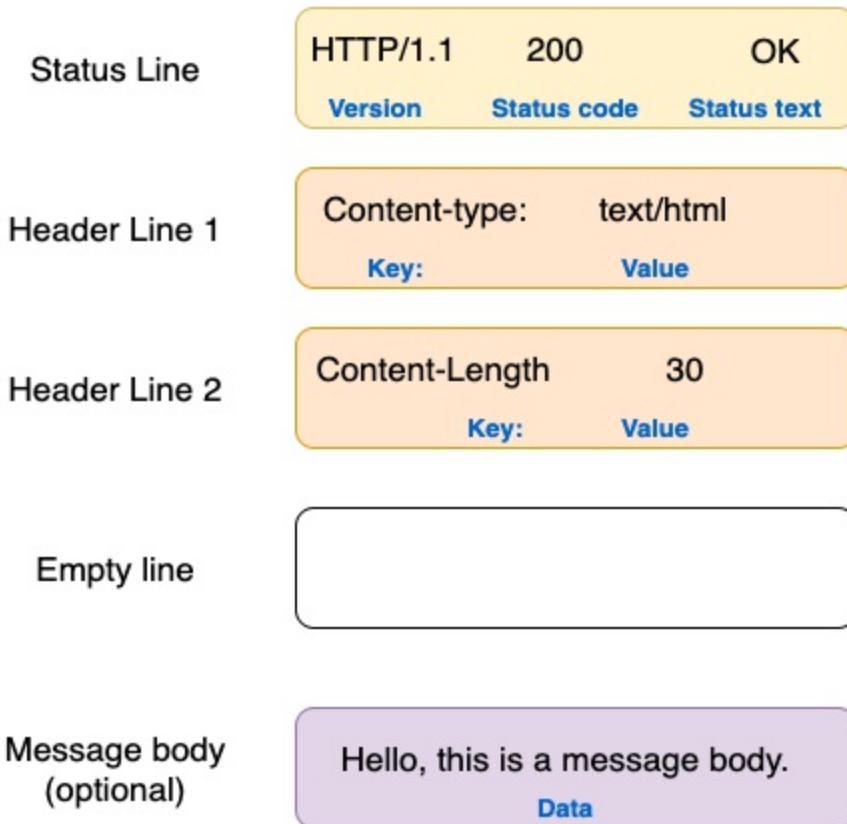
Let's define a struct *HTTPResponse* which will represent the HTTP Response message within our program. We will also write a method to convert this struct (serialize) into a well-formed HTTP message that can be understood by an HTTP client (such as a web browser).

Let's first recap the structure of an HTTP Response message. This will help us define our struct.

Figure 2.7 shows the structure of a typical HTTP response.

Figure 2.7. Structure of HTTP response

Structure of an HTTP Response



First create a file `http/src/httpresponse.rs`, if not created earlier. Add `httpresponse` to the module exports section of `http/lib.rs`, to look like this:

```
pub mod httprequest;
pub mod httpresponse;
```

Add the following code to `http/src/httpresponse.rs`.

Listing 2.11. Structure of HTTP response

```
use std::collections::HashMap;
use std::io::{Result, Write};

#[derive(Debug, PartialEq, Clone)]
pub struct HttpResponse<'a> {
    version: &'a str,
    status_code: &'a str,
```

```
    status_text: &'a str,  
    headers: Option<HashMap<&'a str, &'a str>>,  
    body: Option<String>,  
}
```

The *HttpResponse* struct contains a protocol version, status code, status description, a list of optional headers and an optional body. Note the use of lifetime annotation '*a*' for all the member fields that are of reference types.

Lifetimes in Rust

In Rust, every reference has a lifetime, which is the scope for which the reference is valid. Lifetimes in Rust are an important feature aimed at preventing *dangling pointers* and *use-after-free* errors that are common in languages with manually-managed memory (such as C/C++). The Rust compiler either infers (if not specified) or uses (if specified) the lifetime annotation of a reference to verify that a reference does not outlive the underlying value it points to.

Also note the use of `#[derive]` annotation for traits *Debug*, *PartialEq* and *Clone*. These are called *derivable* traits, because we are asking the compiler to derive the implementation of these traits for our *HttpResponse* struct. By implementing these traits, our struct acquires the ability to be printed out for debugging purposes, can have its member values compared with other values, and have itself cloned.

The list of methods that we will implement for the *HttpResponse* struct is shown here:

1. **Default trait implementation:** We earlier auto-derived a few traits using `#[derive]` annotation. We'll now manually implement the *Default* trait. This lets us specify default values for the struct members.
2. **Method `new()`:** This method creates a new struct with default values for its members.
3. **Method `send_response()`:** This method serializes the contents of the *Http struct* into a valid HTTP response message for on-the-wire transmission, and sends the raw bytes over the TCP connection.
4. **Getter methods:** We'll also implement a set of getter methods for *version*, *status_code*, *status_text*, *headers* and *body*, which are the

member fields of the struct *HttpResponse*.

5. **From trait implementation:** Lastly, we will implement the *From* trait that helps us convert *HttpResponse* struct into a *String* type representing a valid HTTP response message.

Let's add the code for all these under *http/src/httpresponse.rs*.

Default trait implementation

We'll start with the Default trait implementation for *HttpResponse* struct.

Listing 2.12. Default trait implementation for HTTP response

```
impl<'a> Default for HttpResponse<'a> {
    fn default() -> Self {
        Self {
            version: "HTTP/1.1".into(),
            status_code: "200".into(),
            status_text: "OK".into(),
            headers: None,
            body: None,
        }
    }
}
```

Implementing Default trait allows us to do the following to create a new struct with default values:

```
let mut response: HttpResponse<'a> = HttpResponse::default();
```

new() method implementation

The *new()* method accepts a few parameters , sets the default for the others and returns a *HttpResponse* struct. Add the following code under *impl* block of *HttpResponse* struct. As this struct has a reference type for one of its members, the *impl* block declaration has to also specify a lifetime parameter (shown here as '*a*).

Listing 2.13. new() method for HttpResponse (*httpresponse.rs*)

```

impl<'a> HttpResponse<'a> {
    pub fn new(
        status_code: &'a str,
        headers: Option<HashMap<&'a str, &'a str>>,
        body: Option<String>,
    ) -> HttpResponse<'a> {
        let mut response: HttpResponse<'a> = HttpResponse::default();
        if status_code != "200" {
            response.status_code = status_code.into();
        };
        response.headers = match &headers {
            Some(_h) => headers,
            None => {
                let mut h = HashMap::new();
                h.insert("Content-Type", "text/html");
                Some(h)
            }
        };
        response.status_text = match response.status_code {
            "200" => "OK".into(),
            "400" => "Bad Request".into(),
            "404" => "Not Found".into(),
            "500" => "Internal Server Error".into(),
            _ => "Not Found".into(),
        };
        response.body = body;
    }

    response
}
}

```

The *new()* method starts by constructing a struct with default parameters. The values passed as parameters are then evaluated and incorporated into the struct.

send_response() method

The *send_response()* method is used to convert the *HttpResponse* struct into a String, and transmit it over the TCP connection. This can be added within the *impl* block, after the *new()* method in *httpresponse.rs*.

```

impl<'a> HttpResponse<'a> {
    // new() method not shown here
    pub fn send_response(&self, write_stream: &mut impl Write) ->

```

```

        let res = self.clone();
        let response_string: String = String::from(res);
        let _ = write!(write_stream, "{}", response_string);
        Ok(())
    }
}

```

This method accepts a TCP Stream (that implements a Write trait) as input, and writes the well-formed HTTP Response message to the stream.

Getter methods for HTTP response struct

Let's write getter methods for each of the members of the struct. We need these to construct the HTML response message in *httpresponse.rs*.

Listing 2.14. Getter methods for `HttpResponse`

```

impl<'a> HttpResponse<'a> {
    fn version(&self) -> &str {
        self.version
    }
    fn status_code(&self) -> &str {
        self.status_code
    }
    fn status_text(&self) -> &str {
        self.status_text
    }
    fn headers(&self) -> String {
        let map: HashMap<&str, &str> = self.headers.clone().unwr
        let mut header_string: String = "".into();
        for (k, v) in map.iter() {
            header_string = format!("{}{}:{}\r\n", header_string,
        }
        header_string
    }
    pub fn body(&self) -> &str {
        match &self.body {
            Some(b) => b.as_str(),
            None => "",
        }
    }
}

```

The getter methods allow us to convert the data members into string types.

From trait

Lastly, let's implement the method that will be used to convert (`serialize`) `HTTPResponse` struct into an HTTP response message string, in `httpresponse.rs`.

Listing 2.15. Code to serialize Rust struct into HTTP Response message

```
impl<'a> From<HttpResponse<'a>> for String {
    fn from(res: HttpResponse) -> String {
        let res1 = res.clone();
        format!(
            "{} {} {}\r\n{}Content-Length: {}\r\n{}\r\n{}",
            &res1.version(),
            &res1.status_code(),
            &res1.status_text(),
            &res1.headers(),
            &res.body.unwrap().len(),
            &res1.body()
        )
    }
}
```

Note the use of `\r\n` in format string. This is used to insert a new line character. Recall that the HTTP response message consists of the following sequence: status line, headers, blank line and optional message body.

Let's write a few unit tests. Create a test module block as shown and add each test to this block. Don't type this in yet, this is just to show the structure of test code.

```
#[cfg(test)]
mod tests {
    use super::*;

    // Add unit tests here. Each test needs to have a #[test] an
}
```

We'll first check for construction of HTTP response struct for message with status code of 200 (Success).

Add the following to `httpresponse.rs` towards the end of the file.

Listing 2.16. Test script for HTTP success (200) message

```
#[cfg(test)]
mod tests {
    use super::*;

#[test]
fn test_response_struct_creation_200() {
    let response_actual = HttpResponse::new(
        "200",
        None,
        Some("Item was shipped on 21st Dec 2020".into()),
    );
    let response_expected = HttpResponse {
        version: "HTTP/1.1",
        status_code: "200",
        status_text: "OK",
        headers: {
            let mut h = HashMap::new();
            h.insert("Content-Type", "text/html");
            Some(h)
        },
        body: Some("Item was shipped on 21st Dec 2020".into())
    };
    assert_eq!(response_actual, response_expected);
}
}
```

We'll test one for 404 (page not found) HTTP message. Add the following test case *within* the `mod tests {}` block, after the test function `test_response_struct_creation_200()`:

Listing 2.17. Test script for 404 message

```
#[test]
fn test_response_struct_creation_404() {
    let response_actual = HttpResponse::new(
        "404",
        None,
        Some("Item was shipped on 21st Dec 2020".into()),
    );
    let response_expected = HttpResponse {
        version: "HTTP/1.1",
        status_code: "404",
        status_text: "Not Found",
        headers: {
```

```

        let mut h = HashMap::new();
        h.insert("Content-Type", "text/html");
        Some(h)
    },
    body: Some("Item was shipped on 21st Dec 2020".into())
};
assert_eq!(response_actual, response_expected);
}

```

Lastly, we'll check if the HTTP response struct is being serialized correctly into an on-the-wire HTTP response message in the right format. Add the following test *within* the `mod tests {}` block, after the test function `test_response_struct_creation_404()`.

Listing 2.18. Test script to check for well-formed HTTP response message

```

#[test]
fn test_http_response_creation() {
    let response_expected = HttpResponse {
        version: "HTTP/1.1",
        status_code: "404",
        status_text: "Not Found",
        headers: {
            let mut h = HashMap::new();
            h.insert("Content-Type", "text/html");
            Some(h)
        },
        body: Some("Item was shipped on 21st Dec 2020".into())
    };
    let http_string: String = response_expected.into();
    let response_actual = "HTTP/1.1 404 Not Found\r\nContent-
    assert_eq!(http_string, response_actual);
}

```

Let's run the tests now. Run the following from the workspace root:

```
cargo test -p http
```

You should see the following message showing that 6 tests have passed in the `http` module. Note this includes tests for both HTTP request and HTTP response modules.

```
running 6 tests
test httprequest::tests::test_method_into ... ok
```

```
test httprequest::tests::test_version_into ... ok
test httpresponse::tests::test_http_response_creation ... ok
test httpresponse::tests::test_response_struct_creation_200 ... o
test httprequest::tests::test_read_http ... ok
test httpresponse::tests::test_response_struct_creation_404 ... o

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured; 0 fil
```

If the test fails, check for any typos or misalignment in the code (if you had copy pasted it). In particular re-check the following string literal (which is quite long and prone to mistakes):

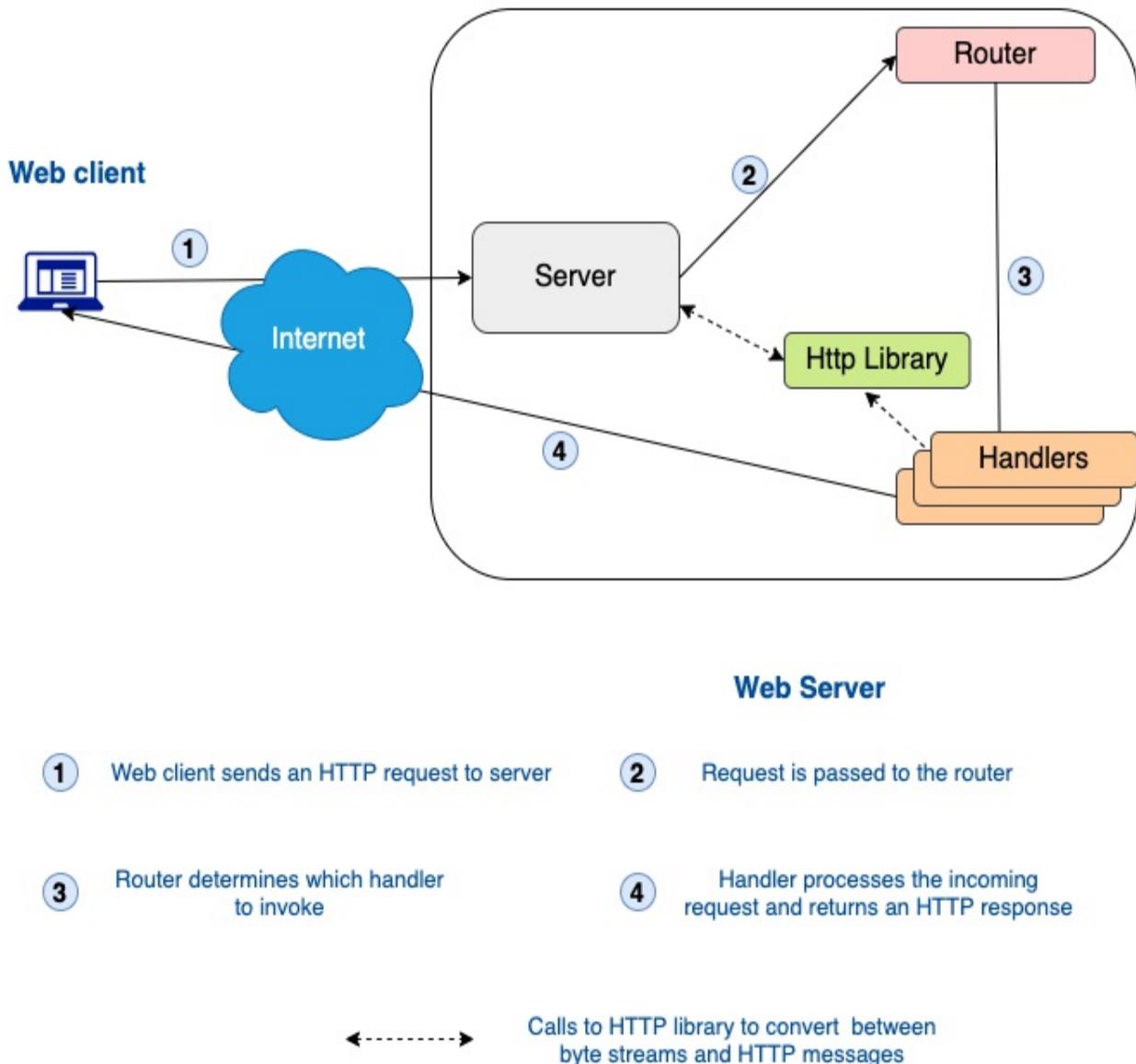
```
"HTTP/1.1 404 Not Found\r\nContent-Type:text/html\r\nContent-Leng
```

If you are still having trouble executing the tests, refer back to the git repo.

This completes the code for the *http* library. Let's recall the design of the http server, shown again in **figure 2.8**.

Figure 2.8. Web server message flow

Web server message flow



We've written the http library. Let's write the `main()` function, `server`, `router` and `handler`. We will have to switch from `http` project to `httpserver` project directory from here on, to write code.

In order to refer to the `http` library from `httpserver` project, add the following to the `Cargo.toml` of the latter.

```
[dependencies]
```

```
http = {path = "../http"}
```

2.2.3 Writing the main() function and server module

Let's take a top-down approach. We'll start with the *main()* function in *httpserver/src/main.rs*:

Listing 2.19. *main()* function

```
mod handler;
mod server;
mod router;
use server::Server;
fn main() {
    // Start a server
    let server = Server::new("localhost:3000");
    //Run the server
    server.run();
}
```

The main function imports three modules - *handler* , *server* and *router*.

Next, create three files - *handler.rs*, *server.rs* and *router.rs* under *httpserver/src*.

Server module

Let's write the code for the server module in *httpserver/src/server.rs*.

Listing 2.20. Server module

```
use super::router::Router;
use http::httprequest::HttpRequest;
use std::io::prelude::*;
use std::net::TcpListener;
use std::str;
pub struct Server<'a> {
    socket_addr: &'a str,
}
impl<'a> Server<'a> {
    pub fn new(socket_addr: &'a str) -> Self {
        Server { socket_addr }
```

```

    }
    pub fn run(&self) {
        // Start a server listening on socket address
        let connection_listener = TcpListener::bind(self.socket_addr);
        println!("Running on {}", self.socket_addr);
        // Listen to incoming connections in a loop
        for stream in connection_listener.incoming() {
            let mut stream = stream.unwrap();
            println!("Connection established");
            let mut read_buffer = [0; 90];
            stream.read(&mut read_buffer).unwrap();
            // Convert HTTP request to Rust data structure
            let req: HttpRequest = String::from_utf8(read_buffer);
            // Route request to appropriate handler
            Router::route(req, &mut stream);
        }
    }
}

```

The server module has two methods:

new() accepts a socket address (host and port), and returns a Server instance.
run() method performs the following:

- binds on the socket,
- listens to incoming connections,
- reads a byte stream on a valid connection,
- converts the stream into an *HttpRequest* struct instance
- Passes the request to *Router* for further processing

2.2.4 Writing the router and handler modules

The *router* module inspects the incoming HTTP request and determines the right handler to route the request to, for processing. Add the following code to *httpserver/src/router.rs*.

Listing 2.21. Router module

```

use super::handler::{Handler, PageNotFoundHandler, StaticPageHandler};
use http::{HttpRequest, HttpResponse};
use std::io::prelude::*;

pub struct Router;

```

```

impl Router {
    pub fn route(req: HttpRequest, stream: &mut impl Write) -> () {
        match req.method {
            // If GET request
            httprequest::Method::Get => match &req.resource {
                httprequest::Resource::Path(s) => {
                    // Parse the URI
                    let route: Vec<&str> = s.split("/").collect()
                    match route[1] {
                        // if the route begins with /api, invoke 'api'
                        "api" => {
                            let resp: HttpResponse = WebServiceHandler::handle();
                            let _ = resp.send_response(stream);
                        }
                        // Else, invoke static page handler
                        _ => {
                            let resp: HttpResponse = StaticPageHandler::handle();
                            let _ = resp.send_response(stream);
                        }
                    }
                },
                // If method is not GET request, return 404 page
                _ => {
                    let resp: HttpResponse = PageNotFoundHandler::handle();
                    let _ = resp.send_response(stream);
                }
            }
        }
    }
}

```

The *Router* checks if the incoming method is a *GET* request. If so, it performs checks in the following order:

- If the *GET* request route begins with */api*, it routes the request to the *WebServiceHandler*
- If the *GET* request is for any other resource, it assumes the request is for a static page and routes the request to the *StaticPageHandler*
- If it is not a *GET* request, it sends back a 404 error page

Let's look at the *Handler* module next.

Handlers

For the handler modules, let's add a couple of external crates to handle json serialization and deserialization - *serde* and *serde_json*. The *Cargo.toml* file for *httpserver* project would look like this:

```
[dependencies]
http = {path = "../http"}
serde = {version = "1.0.117", features = ["derive"]}
serde_json = "1.0.59"
```

Add the following code to *httpserver/src/handler.rs*.

Let's start with module imports:

```
use http::{HttpRequest, HttpResponse};
use serde::{Deserialize, Serialize};
use std::collections::HashMap;
use std::env;
use std::fs;
```

Let's define a trait called *Handler* as shown:

Listing 2.22. Trait Handler definition

```
pub trait Handler {
    fn handle(req: &HttpRequest) -> HttpResponse;
    fn load_file(file_name: &str) -> Option<String> {
        let default_path = format!("{}public", env!("CARGO_MANIF
        let public_path = env::var("PUBLIC_PATH").unwrap_or(defau
        let full_path = format!("{}{}", public_path, file_name);

        let contents = fs::read_to_string(full_path);
        contents.ok()
    }
}
```

Note that the trait *Handler* contains two methods:

- *handle()*: This method has to be implemented for any other user data type to implement the trait.
- *load_file()* : This method is to load a file (non-json) from *public* directory in *httpserver* root folder. The implementation is already provided as part of trait definition.

We'll now define the following data structures:

- **StaticPageHandler** - to serve static web pages,
- **WebServiceHandler** - to serve json data
- **PageNotFoundHandler** - to serve 404 page
- **OrderStatus** - struct used to load data read from json file

Add the following code to *httpserver/src/handler.rs*.

Listing 2.23. Data structures for handler

```
#[derive(Serialize, Deserialize)]
pub struct OrderStatus {
    order_id: i32,
    order_date: String,
    order_status: String,
}

pub struct StaticPageHandler;

pub struct PageNotFoundHandler;

pub struct WebServiceHandler;
```

Let's implement the *Handler* trait for the three handler structs. Let's start with the *PageNotFoundHandler*.

```
impl Handler for PageNotFoundHandler {
    fn handle(_req: &HttpRequest) -> HttpResponse {
        HttpResponse::new("404", None, Self::load_file("404.html"))
    }
}
```

If the handle method on *PageNotFoundHandler* struct is invoked, it would return a new *HttpResponse* struct instance with status code:404, and body containing some html loaded from file *404.html*.

Here is the code for *StaticPageHandler*.

Listing 2.24. Handler to serve static web pages

```
impl Handler for StaticPageHandler {
```

```

fn handle(req: &HttpRequest) -> HttpResponse {
    // Get the path of static page resource being requested
    let http::httprequest::Resource::Path(s) = &req.resource;

    // Parse the URI
    let route: Vec<&str> = s.split("/").collect();
    match route[1] {
        "" => HttpResponse::new("200", None, Self::load_file(
        "health" => HttpResponse::new("200", None, Self::load
        path => match Self::load_file(path) {
            Some(contents) => {
                let mut map: HashMap<&str, &str> = HashMap::n
                if path.ends_with(".css") {
                    map.insert("Content-Type", "text/css");
                } else if path.ends_with(".js") {
                    map.insert("Content-Type", "text/javascript");
                } else {
                    map.insert("Content-Type", "text/html");
                }
                HttpResponse::new("200", Some(map), Some(cont
            }
        }
        None => HttpResponse::new("404", None, Self::load
    },
},
}
}

```

If the *handle()* method is called on the *StaticPageHandler*, the following processing is performed:

- If incoming request is for localhost:3000/, the contents from file *index.html* is loaded and a new *HttpResponse* struct is constructed
- If incoming request is for localhost:3000/health, the contents from file *health.html* is loaded, and a new *HttpResponse* struct is constructed
- If the incoming request is for any other file, the method tries to locate and load that file in the *httpserver/public* folder. If a file is not found, it sends back a 404 error page. If the file is found, the contents are loaded and embedded within an *HttpResponse* struct. Note that the *Content-Type* header in HTTP Response message is set according to the type of file.

Let's look at the last part of the code - *WebServiceHandler*.

Listing 2.25. Handler to serve json data

```
impl WebServiceHandler {
    fn load_json() -> Vec<OrderStatus> { #1
        let default_path = format!("{}{}/data", env!("CARGO_MANIFEST_DIR"), "orders.json");
        let data_path = env::var("DATA_PATH").unwrap_or(default_path);
        let full_path = format!("{}{}/{}", data_path, "orders.json");
        let json_contents = fs::read_to_string(full_path);
        let orders: Vec<OrderStatus> =
            serde_json::from_str(json_contents.unwrap().as_str());
        orders
    }
}

// Implement the Handler trait
impl Handler for WebServiceHandler {
    fn handle(req: &HttpRequest) -> HttpResponse {
        let http::httprequest::Resource::Path(s) = &req.resource;

        // Parse the URI
        let route: Vec<&str> = s.split("/").collect();
        // if route if /api/shipping/orders, return json
        match route[2] {
            "shipping" if route.len() > 2 && route[3] == "orders" =>
                let body = Some(serde_json::to_string(&Self::load_json()));
                let mut headers: HashMap<&str, &str> = HashMap::new();
                headers.insert("Content-Type", "application/json");
                HttpResponse::new("200", Some(headers), body)
            _ => HttpResponse::new("404", None, Self::load_file("orders.html"))
        }
    }
}
```

If `handle()` method is called on the `WebServiceHandler` struct, the following processing is done:

- If the GET request is for `localhost:3000/api/shipping/orders`, the json file with orders is loaded, and this is serialized into json, which is returned as part of the body of the response.
- If it is any other route, a 404 error page is returned.

We're done with the code. We now have to create the html and json files, in order to test the web server.

2.2.5 Testing the web server

In this section, we'll first create the test web pages and json data. We'll then test the web server for various scenarios and analyse the results.

Create two subfolders *data* and *public* under *httpserver* root folder. Under *public* folder, create four files - *index.html*, *health.html*, *404.html*, *styles.css*. Under the *data* folder, create the following file - *orders.json*.

The indicative contents are shown here. You can alter them as per your preference.

httpserver/public/index.html

Listing 2.26. Index web page

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <link rel="stylesheet" href="styles.css">
    <title>Index!</title>
  </head>
  <body>
    <h1>Hello, welcome to home page</h1>
    <p>This is the index page for the web site</p>
  </body>
</html>
```

httpserver/public/styles.css

```
h1 {
  color: red;
  margin-left: 25px;
}
```

httpserver/public/health.html

Listing 2.27. Health web page

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="utf-8" />
  <title>Health!</title>
</head>
<body>
  <h1>Hello welcome to health page!</h1>
  <p>This site is perfectly fine</p>
</body>
</html>
```

httpserver/public/404.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8" /> <title>Not Found!</title>
</head>
<body>
  <h1>404 Error</h1>
  <p>Sorry the requested page does not exist</p>
</body>
</html>
```

httpserver/data/orders.json

Listing 2.28. Json data file for orders

```
[
  {
    "order_id": 1,
    "order_date": "21 Jan 2020",
    "order_status": "Delivered"
  },
  {
    "order_id": 2,
    "order_date": "2 Feb 2020",
    "order_status": "Pending"
  }
]
```

We're ready to run the server now.

Run the web server from the workspace root as shown:

```
cargo run -p httpserver
```

Then from either a browser window or using *curl* tool, test the following URLs:

```
localhost:3000/  
localhost:3000/health  
localhost:3000/api/shipping/orders  
localhost:3000/invalid-path
```

You'll notice that if you invoke these commands on the browser, for the first URL you should see the heading in red font. Go to *network* tab in chrome browser (or equivalent dev tools on other browsers) and view the files downloaded by browser. You'll see that in addition to the *index.html* file, the *styles.css* is also automatically downloaded by the browser which results in the styling applied to the index page. If you inspect further, you can see that *Content-Type* of *text/css* has been sent for the *css* file and *text/html* has been send for the *HTML* file, from our web server to the browser.

Likewise, if you inspect the response *content-type* sent for */api/shipping/orders* path, you will see *application/json* received by the browser as part of response headers.

This concludes the section on building a web server.

In this section, we have written an HTTP server and a library of http messages that can serve static pages, as well as serve json data. While the former capability is associated with the term web server, the latter is where we start to see web service capabilities. Our *httpserver* project functions as both a *static web server* as well as a *web service* serving json data. Of course, a regular *web service* would serve more methods than just GET requests. But this exercise was intended to demonstrate capabilities of Rust to build such a *web server* and *web service* from scratch, without using any web frameworks or external http libraries.

I hope you enjoyed following along the code, and got to a working server. If you have any difficulties, you can refer back to the code repository for chapter 2.

This brings an end to the two core objectives of the chapter, viz to *build a TCP server/client* and to *build an HTTP server*.

The complete code for this chapter can be found at
<https://git.manning.com/agileauthor/eshwarla/-/tree/master/code>.

2.3 Summary

- The TCP/IP model is a simplified set of standards and protocols for communication over the internet. It is organized into four abstract layers: Network Access layer, Internet Layer, Transport Layer and the Application layer. TCP is the *transport-layer* protocol over which other *application-level* protocols such as HTTP operate. We built a server and client that exchanged data using the TCP protocol.
- TCP is also a *stream-oriented* protocol where data is exchanged as a continuous stream of bytes.
- We built a basic TCP server and client using the Rust standard library. TCP does not understand the semantics of messages such as HTTP. Our TCP client and server simply exchanged a stream of bytes without any understanding of the semantics of data transmitted.
- HTTP is an application layer protocol and is the foundation for most web services. HTTP uses TCP in most cases as the transport protocol.
- We built an HTTP library to parse incoming HTTP requests and construct HTTP responses. The HTTP requests and responses were modeled using Rust *structs* and *enums*.
- We built an HTTP server that serves two types of content - *static web pages* (with associated files such as stylesheets), and *json data*.
- Our web server can accept requests and send responses to standard HTTP clients such as browsers and *curl* tool.
- We added additional behaviour to our custom structs by implementing several traits. Some of them were auto-derived using Rust annotations, and others were hand-coded. We also made use of lifetime annotations to specify lifetimes of references within structs.

You now have the foundational knowledge to understand how Rust can be used to develop a low-level HTTP library and web server, and also beginnings of a web service. In the next chapter we will dive right into developing web services using a production-ready web framework that is written in Rust.

3 Building a RESTful Web Service

This chapter covers

- Getting started with Actix
- Writing a RESTful web service

In this chapter, we will build our first real web service.

The web service will expose a set of APIs over HTTP, and will use the **Representational State Transfer (REST)** architectural style.

We'll build the web service using **Actix**, a lightweight web framework written in Rust, which is also one of the most mature in terms of code activity, adoption and ecosystem. We will warm-up by writing introductory code in Actix to understand its foundational concepts and structure. Later, we will design and build a set of REST APIs using an in-memory data store that is thread-safe.

The complete code for this chapter can be found at
<https://git.manning.com/agileauthor/eshwarla/-/tree/master/code>.

Let's get started.

Why Actix?

This book is about developing high performance web services and applications in Rust. The web frameworks considered while writing this book were Actix, Rocket, Warp and Tide. While Warp and Tide are relatively newer, Actix and Rocket lead the pack in terms of adoption and level of activity. Actix was chosen over Rocket as Rocket does not yet have native async support, and async support is a key factor to improve performance in I/O-heavy workloads (such as web service apis) at scale.

3.1 Getting started with Actix

In this book, you are going to build a *digital storefront aimed at tutors*.

Let's call our digital platform **EzyTutors**, because we want tutors to easily publish their training catalogs online, which can trigger the interest of learners and generate sales.

To kickstart this journey, we'll build a set of simple APIs that allow tutors to create a course and learners to retrieve courses for a tutor.

This section is organised into two parts. In the first section, we will build a basic async HTTP server using Actix that demonstrates a simple health-check API. This will help you understand the foundational concepts of Actix. In the second section, we will design and build REST APIs for the *tutor* web service. We will rely on an in-memory data store (rather than a database) and use test-driven development. Along the way, you will be introduced to key Actix concepts such as *routes*, *handlers*, *HTTP request* parameters and *HTTP responses*.

Let's write some code, shall we?

3.1.1 Writing the first REST API

In this section we'll write our first Actix server, which can respond to an HTTP request.

A note about the environment

There are many ways to organize code that you will be building out over the course of this book.

The first option is to create a workspace project (similar to the one we created in Chapter 2), and create separate projects under the workspace, one per chapter.

The second option is to create a separate cargo binary project for each chapter. Grouping options for deployment can be determined at a later time.

Either approach is fine, but in this book, we will adopt the first approach to

keep things organised together. We'll create a workspace project - **ezytutors** which will hold other projects.

Create a new project with

```
cargo new ezytutors && cd ezytutors
```

This will create a *binary* cargo project. Let's convert this into a *workspace* project. Under this *workspace*, let's store the web service and web applications that we will build in future chapters.

Add the following to *Cargo.toml*:

```
[workspace]
members = ["tutor-nodb"]
```

tutor-nodb will be the name of the webservice we will be creating in this chapter. Create another cargo project as follows:

```
cargo new tutor-nodb && cd tutor-nodb
```

This will create a *binary* Rust project called **tutor-nodb** under the **ezytutors** workspace. For convenience, we will call this *tutor web service* henceforth. The root folder of this cargo project contains *src* subfolder and *Cargo.toml* file.

Add the following dependencies in *Cargo.toml* of course web service:

```
[dependencies]
actix-web = "4.2.1"          #1
actix-rt = "2.7.0"            #2
```

Add the following binary declaration to the same *Cargo.toml* file, to specify the name of the binary file.

```
[[bin]]
name = "basic-server"
```

Let's now create a source file called *basic-server.rs* under the *tutor-nodb/src/bin* folder. This will contain the *main()* function which is the entry point for the binary.

There are four basic steps to create and start a basic HTTP server in Actix:

- Configure *routes*: Routes are paths to various resources in a web server. For our example, we will configure a route */health* to do health checks on the server.
- Configure *handler*: Handler is the function that processes requests for a route. We will define a *health-check handler* to service the */health* route.
- Construct a *web application* and register routes and handlers with the application.
- Construct an *HTTP server* linked to the *web application* and run the server.

These four steps are shown in the code with annotations. Add the following code to *src/bin/basic-server.rs*. Don't worry if you don't understand all the steps and code, just type it in for now, and they will be explained in detail later.

Note: I would highly recommend that you type in the code line-by-line rather than copy and paste it into your editor. This will provide a better return on your investment of time in learning, as you will be practising rather than just reading.

Listing 3.1. Writing a basic Actix web server

```
// Module imports
use actix_web::{web, App, HttpResponse, HttpServer, Responder};
use std::io;

// Configure route #1
pub fn general_routes(cfg: &mut web::ServiceConfig) {
    cfg.route("/health", web::get().to(health_check_handler));
}

//Configure handler #2
pub async fn health_check_handler() -> impl Responder {
    HttpResponse::Ok().json("Hello. EzyTutors is alive and kickin")
}

// Instantiate and run the HTTP server
#[actix_rt::main]
async fn main() -> io::Result<()> {
```

```
// Construct app and configure routes #3
let app = move || App::new().configure(general_routes);

// Start HTTP server #4
HttpServer::new(app).bind("127.0.0.1:3000")?.run()?.await
}
```

You can run the server in one of two ways.

If you are in the *ezytutors* workspace folder root, run the following command:

```
cargo run -p tutor-nodb --bin basic-server
```

The *-p* flag tells cargo tool to build and run the binary for project *tutor-nodb*, within the workspace.

Alternatively, you can run the command from within the *tutor-nodb* folder as follows:

```
cargo run --bin basic-server
```

In a web browser window, visit the following URL:

```
localhost:3000/health
```

You will see the following printed:

```
Hello, EzyTutors is alive and kicking
```

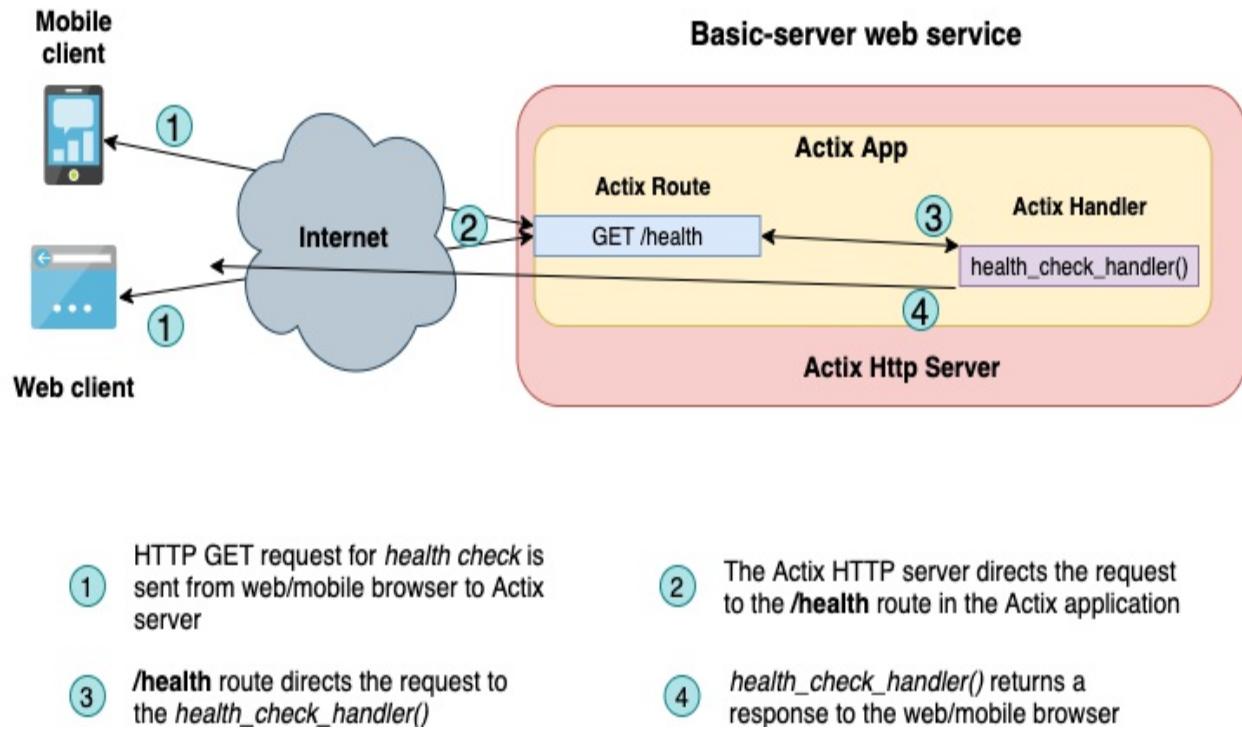
Congratulations! You have built your first REST API in Actix.

3.1.2 Understanding Actix concepts

In the previous section, we wrote a basic Actix web server (aka Actix HTTP server). The server was configured to run a web application with a single route */health* which returns the health status of the web application service. **Figure 3.1** shows the various components of Actix that we used in the code.

Figure 3.1. Actix Basic server

Components of the Actix web service



Here is the sequence of steps:

1. When you typed *localhost:3000/health* in your browser, an *HTTP GET* request message was constructed by the browser, and sent to the Actix *basic-server* listening at *localhost:3000* port.
2. The Actix *basic-server* inspected the GET request and determined the route in the message to be */health*. The basic server then routed the request to the web application (App) that has the */health* route defined.
3. The web application in turn determined the handler for the route */health* to be *health_check_handler()* and routed the message to the handler.
4. The *health_check_handler()* constructs an HTTP response with a text message and sends it back to the browser.

You would have noticed the terms *HTTP server*, *Web Application*, *Route* and *Handler* used prominently. These are key concepts within Actix to build web services. Recall that we used the terms server, route and handler also in

chapter 2. Conceptually, these are similar. But let us understand them in more detail in the context of *Actix*.

HTTP (web) Server: It is responsible for serving HTTP requests. It understands and implements the HTTP protocol. By default, the HTTP server starts a number of threads (called workers) to process incoming requests.

Actix concurrency

Actix supports two levels of concurrency. It supports asynchronous I/O wherein a given os-native thread performs other tasks while waiting on I/O (such as listening for network connections). It also supports multi-threading for parallelism, and starts a number of OS-native threads (called *workers*) equal to the number of logical CPUs in the system, by default.

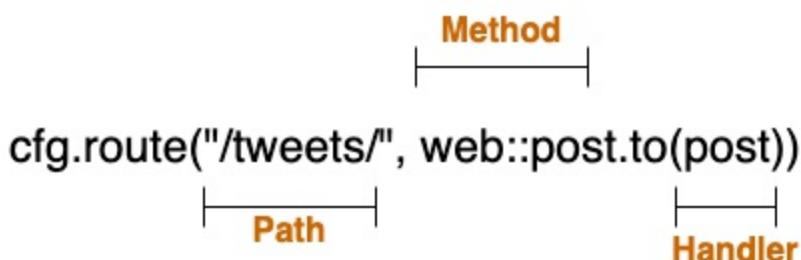
Actix HTTP server is built around the concept of web applications and requires one for initialization. It constructs an application instance per OS thread.

App: This represents an Actix web application. An Actix web application is a grouping of the set of routes it can handle.

Routes and handlers A route in Actix tells the Actix web server how to process an incoming request.

A route is defined in terms of a *route path*, an *HTTP method* and a *handler* function. Said differently, a request handler is registered with an application's *route* on a *path* for a particular *HTTP method*. The structure of an Actix route is illustrated in figure here.

Figure 3.2. Structure of Actix route



This is the route we implemented earlier for health check:

```
cfg.route(  
    "/health", #1  
    web::get() #2  
    .to(health_check_handler)); #3
```

The route shown above specifies that if a *GET HTTP* request arrives for the path */health*, the request should be routed to the request handler method *health_check_handler()*.

A request handler is an asynchronous method that accepts zero or more parameters and returns an HTTP response.

The following is a request handler that we implemented in the previous example.

```
pub async fn health_check_handler() -> impl Responder {  
    HttpResponse::Ok().json("Hello, EzyTutors is alive and kickin  
}
```

In code shown, *health_check_handler()* is a function that implements *Responder* trait. Types that implement *Responder* trait acquire the capability to send HTTP responses. Note that our handler does not accept any input parameter, but it is possible to send data along with HTTP requests from the client, that will be made available to handlers. We'll see such an example in the next section.

More about Actix-web

Listed here are a few more details about the Actix web framework.

Actix-web is a modern, rust-based, light-weight and fast web framework. Actix-web has consistently featured among the best web frameworks in TechEmpower performance benchmarks, which can be found here: <https://www.techempower.com/benchmarks/>. Actix-web is among the most mature Rust web frameworks and supports several features as listed here:

- Support for HTTP/1.x and HTTP/2
- Support for request and response pre-processing

- Middleware can be configured for features such as CORS, session management, logging, and authentication
- It supports asynchronous I/O. This provides the ability for the Actix server to perform other activities while waiting on network I/O.
- Content compression
- Can connect to multiple databases
- Provides an additional layer of testing utilities (over the Rust testing framework) to support testing of HTTP requests and responses
- Supports static web page hosting and server-rendered templates

More technical details about the Actix web framework can be found here:
<https://docs.rs/crate/actix-web/2.0.0>

Using a framework like Actix-Web significantly speeds up the time for prototyping and development of web APIs in Rust, as it takes care of the low-level details of dealing with HTTP protocols and messages, and provides several utility functions and features to make web application development easier.

While Actix-web has an extensive feature set, we'll be able to cover only a subset of the features in this book. The features that we'll cover include HTTP methods that provide CRUD (Create-Read-Update-Delete) functionality for resources, persistence with databases, error handling, state management, JWT authentication, and configuring middleware.

In this section, we built a basic Actix web service exposing a health check API, and reviewed key features of the Actix framework. In the next section, we will build the web service for the *EzyTutors* social network.

3.2 Building web APIs with REST

This section will take you through the typical steps in developing a RESTful web service with Actix.

A web service is a network-oriented service. Network-oriented services communicate through messages over a network. Web services use HTTP as the primary protocol for exchanging messages. There are several architectural

styles that can be used to develop web services such as SOAP/XML, REST/HTTP and gRPC/HTTP. In this chapter we will use the REST architectural style.

REST APIs

REST stands for *Representational State transfer*. It is a term used to visualize web services as a network of resources each having its own state. Users trigger operations such as GET, PUT , POST or DELETE on resources identified by URIs (for example, <https://www.google.com/search?q=weather%20berlin> can be used to get the current weather at Berlin). Resources are application entities such as users, shipments, courses etc. Operations on resources such as POST and PUT can result in *state changes* in the resources. The latest state is returned to the client making the request.

REST architecture defines a set of properties (called constraints) that a web service must adopt, and are listed below:

- *Client-server architecture* for separation of concerns, so client and server are decoupled and can evolve independently.
- *Statelessness*: Stateless means there is no client context stored on the server between consecutive requests from the same client.
- *Layered system*: Allows the presence of intermediaries such as load balancers and proxies between the client and the server.
- *Cacheability*: Supports caching of server responses by clients to improve performance.
- *Uniform interface*: Defines uniform ways to address and manipulate resources, and to standardize messages.
- *Well defined state changes*: For example, GET requests do not result in state change, but POST, PUT and DELETE messages do.

Note that REST is not a formal standard, but an architectural style. So, there may be variations in the way RESTful services are implemented.

A web service that exposes APIs using the REST architectural style is called a RESTful web service. We'll build a RESTful web service in this section for our *EzyTutors* digital storefront. We've chosen the RESTful style for the APIs because they are intuitive, widely used, and suited for external-facing

APIs (as opposed to say, gRPC which is more suited to APIs between internal services).

The core functionality of our web service in this chapter will be to allow *posting of a new course*, *retrieving course list for a tutor*, and *retrieving details for an individual course*. Our initial data model will contain just one resource: *course*. But before getting to the data model, let's finalize the structure of the project and code organization, and also determine how to store this data in memory in a way that is safely accessible across multiple Actix worker threads.

3.2.1 Define project scope and structure

In this section, let's define the scope of what we'll be building, and how code will be organised within the project.

We will build three RESTful APIs for the *tutor* web service. These APIs will be registered on an *Actix web application*, which in turn will be deployed on the *Actix HttpServer*.

The APIs are designed to be invoked from a web front-end or mobile application. We'll test the GET API requests using a standard browser, and the POST request using *curl*, a command-line HTTP client (you can also use a tool like Postman, if you prefer).

We'll use an in-memory data structure to store courses, instead of a database. This is just for simplicity. A relational database will be added in the next chapter.

Figure 3.3 shows the various components of the Web service that we'll be building.

Figure 3.3. Components of the web service

RESTful APIs with Actix

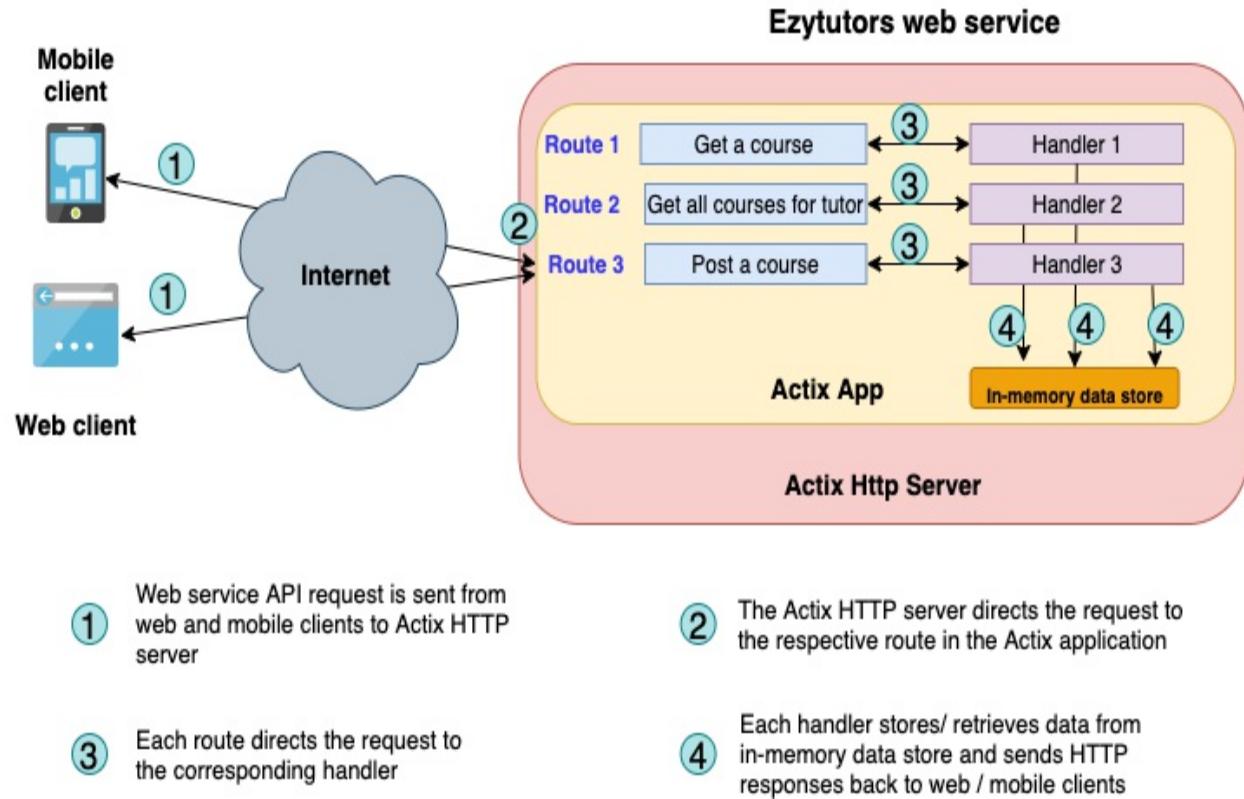


Figure 3.3 shows how the HTTP requests from web and mobile clients are handled by the web service. Recall a similar figure we saw for the *basic-server* in the previous section. Here is the sequence of steps in the request and response message flow:

1. The HTTP requests are constructed by web or mobile clients and sent to the domain address and port number where the *Actix web server* is listening.
2. The *Actix web server* routes the request to the *Actix web app*.
3. The *actix web app* has been configured with the routes for the three APIs. It inspects the route configuration, determines the right handler for the specified route, and forwards the request to the *handler* function.
4. The request *handlers* parse the request parameters, read or write to the in-memory data store, and return an HTTP response. Any errors in processing are also returned as HTTP responses with the appropriate

status codes.

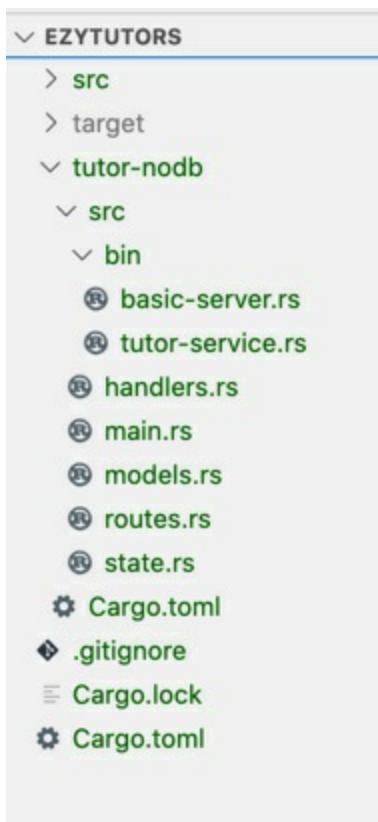
This, in brief, is how a request-response flow works in Actix web.

Here is a closer look at the APIs that we will build:

1. **POST /courses**: Create a new course and save it in the webservice.
2. **GET /courses/tutor_id**: Get a list of courses offered by a tutor
3. **GET /courses/tutor_id/course_id**: Get course details

We have reviewed the scope of the project. We can now take a look at how the code will be organised. **Figure 3.4** shows the code structure.

Figure 3.4. Project structure of EzyTutors web service



Here is the structure of the project:

1. **bin/tutor-service.rs** Contains the main() function
2. **models.rs** Contains the data model for the web service

3. **state.rs** Application state is defined here
4. **routes.rs** Contains the route definitions
5. **handlers.rs** Contains handler functions that respond to HTTP requests
6. **Cargo.toml** Configuration file and dependencies specification for the project

Next, update the *Cargo.toml* to look like this:

Listing 3.2. Configuration for the Basic Actix web server

```
[package]
name = "tutor-nodb"
version = "0.1.0"
authors = ["peshwar9"]
edition = "2018"
default-run="tutor-service"

[[bin]]
name = "basic-server"

[[bin]]
name = "tutor-service"

[dependencies]
#Actix web framework and run-time
actix-web = "3.0.0"
actix-rt = "1.1.1"
```

You will notice that we've defined two binaries for this project. The first one is *basic-server* which we built in the previous section. The second one is *tutor-service* which we will build now.

We also have two dependencies to include - *actix-web* framework and *actix-runtime*.

Note also that under the [package] tag, we've added a parameter *default-run* with a value *tutor_service*. This tells *cargo* that by default the *tutor_service* binary should be built unless otherwise specified. This allows us to build and run the tutor service with *cargo run -p tutor-nodb*, rather than *cargo run -p tutor-nodb --bin tutor-service*.

Create a new file, `tutor-nodb/src/bin/tutor-service.rs`. This will contain the code for the web service in this section.

We've covered the project scope and structure. Let's turn our attention to another topic - how we will store the data in the web service. We've already said we don't want to use a database, but want to store data in memory. This is fine in case of a single-threaded server, like the one we built in the last chapter. But Actix is a multi-threaded server. Each thread (Actix worker) runs a separate instance of the application. How can we make sure that two threads are not trying to mutate the data in-memory simultaneously. Of course, Rust has features such as *Arc* and *Mutex* that we can use to address this problem. But then, where in the web service should we define the shared data, and how can we make this available to the handlers where the processing will take place? Actix Web framework gives us a way to address in an elegant way. Actix allows us to define application state of any custom type, and access it using a built-in extractor. Let's take a closer look at this in the next section.

3.2.2 Define and manage application state

The term *application* state can be used in different contexts to mean different things.

W3C defines application state (reference link here: <https://www.w3.org/2001/tag/doc/state.html>) as how an application is: its configuration, attributes, condition or information content. State changes happen in an application component when triggered by an event. More specifically, in the context of applications that provide a RESTful web API to manage resources over a URI (such as the one we're discussing in this chapter), application state is closely related to the state of the resources that are part of the application. In this chapter we are specifically dealing with *course* as the only resource. So, it can be said that the state of our application changes as courses are added or removed for a tutor. In most real-world applications, the state of resources is persisted to a data store. However, in our case, we will be storing the application state in memory.

Actix web server spawns a number of threads by default, on startup (this is

configurable). Each thread runs an instance of the web application and can process incoming requests independently. However, by design, there is no built-in sharing of data across Actix threads. You may wonder why we would want to share data across threads? Take an example of a database connection pool. It makes sense for multiple threads to use a common connection pool to handle database connections. Such data can be modeled in actix as *Application state*. This state is *injected* by Actix framework into the request handlers such that the handler can access state as a parameters in their method signatures. All routes within an Actix app can share application state.

Why do we want to use application state for the *tutor web service*?

Because we want to store a list of courses in memory as application state. We'd like this state to be made available to all the handlers and shared safely across different threads. But before we go to courses, let's try a simpler example to learn how to define and use application state with Actix.

Let's define a simple application state type with two elements - a *string* data type (representing a static string response to health check request) and an *integer* data type (representing the number of times a user has visited a particular route).

The *string* value will be *shared immutable state* accessible from all threads, i.e the values cannot be modified after initial definition.

The *number* value will be *shared mutable state*, i.e, the value can be mutated from every thread. However, before modifying value, the thread has to acquire control over the data. This is achieved by defining the *number* value with protection of *Mutex*, a mechanism provided in Rust standard library for safe cross-thread communications.

Here is the plan for the first iteration of the tutor-service.

- Define application state for health check API in *src/state.rs*,
- Update the main function (of Actix server) to initialize and register application state in *src/bin/tutor-service.rs*,
- Define the route for healthcheck route in *src/routes.rs*
- Construct HTTP response in *src/handlers.rs* using this application state.

Define application state

Add the following code for application state in *tutor-nodb/src/state.rs*.

```
use std::sync::Mutex;

pub struct AppState {
    pub health_check_response: String,          #1
    pub visit_count: Mutex<u32>,                #2
}
```

Initialize and register application state

Add the following code in *tutor-nodb/src/bin/tutor-service.rs*

Listing 3.3. Building an Actix web server with application state

```
use actix_web::{web, App, HttpServer};
use std::io;
use std::sync::Mutex;

#[path = "../handlers.rs"]
mod handlers;
#[path = "../routes.rs"]
mod routes;
#[path = "../state.rs"]
mod state;

use routes::*;

use state::AppState;

#[actix_rt::main]
async fn main() -> io::Result<()> {
    let shared_data = web::Data::new(AppState {           #1
        health_check_response: "I'm good. You've already asked me ".t,
        visit_count: Mutex::new(0),
    });
    let app = move || {                                #2
        App::new()
            .app_data(shared_data.clone())           #3
            .configure(routes::general_routes)       #4
    };
    HttpServer::new(app).bind("127.0.0.1:3000")?.run().await #5
}
```

```
}
```

Define route

Let's define the health check route in *tutor-nodb/src/routes.rs*.

```
use super::handlers::*;
use actix_web::web;

pub fn general_routes(cfg: &mut web::ServiceConfig) {
    cfg.route("/health", web::get().to(health_check_handler));
}
```

Update health check handler to use application state

Add the following code for health check handler in *tutor-nodb/src/handlers.rs*

Listing 3.4. Health check handler using application state

```
use super::state::AppState;
use actix_web::{web, HttpResponse};

pub async fn health_check_handler(app_state: web::Data<AppState>)
{
    let health_check_response = &app_state.health_check_response;
    let mut visit_count = app_state.visit_count.lock().unwrap();
    let response = format!("{} {} times", health_check_response,
        *visit_count += 1); #5
    HttpResponse::Ok().json(&response)
}
```

To recap, we

- defined app state in *src/state.rs*,
- registered app state with the Web application in *src/bin/tutor-service.rs*,
- defined the route in *src/routes.rs*, and
- wrote a health check handler function to read and update application state in *src/handlers.rs*.

From the root directory of tutor web service (i.e. *ezytutors/tutor-nodb*), run the following command:

```
cargo run
```

Note that since we have mentioned the default binary in Cargo.toml as shown here, *cargo* tool runs the *course-service* binary by default:

```
default-run="tutor-service"
```

Otherwise, we would have had to specify the following command to run the *tutor-service* binary, as there are two binaries defined in this project.

```
cargo run --bin tutor-service
```

Go to a browser, and type the following in the URL window:

```
localhost:3000/health
```

Every time you refresh the browser window, you will find the visit count being incremented. You'll see a message similar to this:

```
I'm good. You've already asked me 2 times
```

We've so far seen how to define and use application state. This is quite a useful feature for sharing data and injecting dependencies across the application in a safe manner. We'll use more of this feature in the coming chapters.

3.2.3 Defining the data model

Before we develop the individual APIs for the tutor web service, let's first take care of two things:

- Define the data model for the web service
- Define the in-memory data store.

These are pre-requisites to build APIs.

Defining the data model for courses

Let's define a Rust data structure to represent a course. A course in our web

application will have the following attributes:

- **Tutor id:** Denotes the tutor who offers the course.
- **Course id:** This is a unique identifier for the course. In our system, a course id will be unique for a tutor.
- **Course name:** This is the name of the course offered by tutor
- **Posted time:** Timestamp when the course was recorded by the web service.

For creating a new course, the user (of the API) has to specify the *tutor_id* and *course_name*. The *course_id* and *posted_time* will be generated by the web service.

We have kept the data model simple, in order to retain focus on the objective of the chapter. For recording *posted_time*, we will use a third-party *crate* (a library is called a *crate* in Rust terminology) *chrono*.

For serializing and deserializing Rust data structures to on-the-wire format (and vice versa) for transmission as part of the HTTP messages, we will use another third-party crate, *serde*.

Let's first update the *Cargo.toml* file in the folder *ezytutor/tutor-nodb*, to add the two external crates - *chrono* and *serde*.

```
[dependencies]
//actix dependencies not shown here

# Data serialization library
serde = { version = "1.0.110", features = ["derive"] }
# Other utilities
chrono = {version = "0.4.11", features = ["serde"]}
```

Add the following code to *tutor-nodb/src/models.rs*.

Listing 3.5. Data model for courses

```
use actix_web::web;
use chrono::NaiveDateTime;
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize, Debug, Clone)] #1
```

```

pub struct Course {
    pub tutor_id: i32,
    pub course_id: Option<i32>,
    pub course_name: String,
    pub posted_time: Option<NaiveDateTime>,           #2
}
impl From<web::Json<Course>> for Course {          #3
    fn from(course: web::Json<Course>) -> Self {
        Course {
            tutor_id: course.tutor_id,
            course_id: course.course_id,
            course_name: course.course_name.clone(),
            posted_time: course.posted_time,
        }
    }
}

```

In the code shown, you will notice that *course_id* and *posted_time* have been declared to be of type `Option<i32>` and `Option<NaiveDateTime>` respectively. What this means is that these two fields can either hold a valid value of type `i32` and `chrono::NaiveDateTime` respectively, or they can both hold a value of `None` if no value is assigned to these fields.

Further, in the code statement annotated by `<3>`, you will notice a *From* trait implementation. This is a trait implementation that contains a function to convert `web::Json<Course>` to `Course` data type. What exactly does this mean?

We earlier saw that application state that is registered with the Actix web server is made available to handlers using the extractor `web::Data<T>`. Likewise, data from incoming request body is made available to handler functions through the extractor `web::Json<T>`. When a POST request is sent from a web client with the *tutor_id* and *course_name* as data payload, these fields are automatically extracted from `web::Json<T>` Actix object and converted to `Course` Rust type, by this method. This is the purpose of the *From* trait implementation in code *listing 3.5*.

Derivable traits

Traits in Rust are like *interfaces* in other languages. They are used to define shared behaviour. Data types implementing a *trait* share common behaviour

that is defined in the *trait*. For example, we can define a *trait* called *RemoveCourse* as shown.

```
trait RemoveCourse {
    fn remove(self, course_id) -> Self;
}
struct TrainingInstitute;
struct IndividualTutor;

impl RemoveCourse for IndividualTutor {
    // An individual tutor's request is enough to remove a course
}
impl RemoveCourse for TrainingInstitute {
    // There may be additional approvals needed to remove a course
}
```

Assuming we have two types of tutors - *training institutes* (business customers) and *individual tutors*, both types can implement the *RemoveCourse* trait (which means they share a common behaviour that courses offered by both types can be removed from our web service). However the exact details of processing needed for removing a course may vary because business customers may have multiple levels of approvals before a decision on removing a course is taken. This is an example of a custom trait. The Rust standard library itself defines several traits, which are implemented by the types within Rust. Interestingly, these traits can be implemented by custom structs defined at the application-level. For example, *Debug* is a trait defined in the Rust standard library to print out value of a Rust data type for debugging. A custom struct (defined by application) can also choose to implement this trait to print out values of the custom type for debugging. Such trait implementations can be auto-derived by the Rust compiler when we specify the `#[derive()]` annotation above the type definition. Such traits are called *derivable traits*. Examples of derivable traits in Rust include *Eq*, *PartialEq*, *Clone*, *Copy* and *Debug*.

Note that such trait implementations can also be manually implemented, if complex behaviour is desired.

Adding course collection to application state

We have defined the data model for *course*. Now, how will we store courses

as they are added?

We do not want to use a relational database or a similar persistent data store. So, let's start with a simpler option.

We earlier saw that Actix provides the feature to share application state across multiple threads of execution. Why not use this feature for our in-memory data store?

We had earlier defined an AppState struct in *tutor-nodb/src/state.rs* to keep track of visit counts. Let's enhance that struct to also store the course collection.

```
use super::models::Course;
use std::sync::Mutex;
pub struct AppState {
    pub health_check_response: String,
    pub visit_count: Mutex<u32>,
    pub courses: Mutex<Vec<Course>>,           #1
}
```

Since we have altered the definition of application state, we should reflect this in main() function.

In *tutor-nodb/src/bin/tutor-service.rs*, make sure that all the module imports are correctly declared.

Listing 3.6. Module imports for main() function

```
use actix_web::{web, App, HttpServer};
use std::io;
use std::sync::Mutex;

#[path = "../handlers.rs"]
mod handlers;
#[path = "../models.rs"]
mod models;
#[path = "../routes.rs"]
mod routes;
#[path = "../state.rs"]
mod state;
use routes::*;


```

```
use state::AppState;
```

Then, in `main()` function, initialize courses collection with an empty vector collection in `AppState`.

```
async fn main() -> io::Result<()> {
    let shared_data = web::Data::new(AppState {
        health_check_response: "I'm good. You've already asked me ".t,
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),           #1
    });
    // other code
}
```

While we haven't written any new API yet, we have done the following:

- Added a data model module,
- Updated the `main()` function, and
- Changed application state struct to include a course collection
- Updated routes and handlers
- Updated `Cargo.toml`

Let's ensure that nothing is broken. Build and run the code with the following command from within the `tutor-nodb` folder:

```
cargo run
```

You should be able to test with the following URL from the web browser, and things should work as before:

```
curl localhost:3000/health
```

If you are able to view the health page with message containing visitor count, you can proceed ahead. If not, review the code in each of the files for oversight or typos. If you still cannot get it to work, refer to the completed code within the code repository.

We're now ready to write the code for the three course-related APIs in the coming sections.

For writing the APIs, let's first define a uniform set of steps that we can

follow (like a template). We will execute these steps for writing each API. By the end of this chapter, these steps should become ingrained in you.

- Step 1: Define the route configuration
- Step 2: Write the handler function
- Step 3: Write automated test scripts
- Step 4: Build the service and test the API

The route configuration for all new routes will be added in *tutor-nodb/src/routes.rs* and the handler function will be added in *tutor-nodb/src/handlers.rs*. The automated test scripts will also be added under *tutor-nodb/src/handlers.rs* for our project.

3.2.4 Post a course

Let's now write the code to implement a REST API for posting a new course. We'll follow the set of steps defined towards the end of the previous section, to implement the API.

Step 1: Define the route configuration

Let's add the following route to *tutor-nodb/src/routes.rs*, after the *general_routes* block:

```
pub fn course_routes(cfg: &mut web::ServiceConfig) {  
    cfg  
        .service(web::scope("/courses")  
            .route("/", web::post().to(new_course)));  
}
```

The expression *service(web::scope("/courses"))* creates a new resource *scope* called *courses*, under which all APIs related to courses can be added. A *scope* is a set of resources with a common root path. A set of routes can be registered under a scope. Application state can be shared among routes within the same scope. For example, we can create two separate scope declarations, one for *courses* and one for *tutors*, and access routes registered under them as follows.

```
localhost:3000/courses/1 // retrieve details for course with id
```

```
localhost:3000/tutors/1 // retrieve details for tutor with id 1
```

These are only examples for illustration, but don't test them yet as we have not yet defined these routes. What we have defined so far is one route under *courses* which matches an incoming *POST* request with path */courses/* and routes it to handler called *new_course*.

Let's look at how we can invoke the route, *after* implementing the API. The command shown next could be used to post a new course.

```
curl -X POST localhost:3000/courses/ -H "Content-Type: application
```

Note, this command will not work yet, because we have to do two things. First, we have to register this new route group with the web application that is initialized in the *main()* function. Secondly, we have to define the *new_course* handler method.

Modify the *main()* function in *tutor-nodb/src/bin/tutor-service.rs* to look like this.

```
let app = move || {
    App::new()
        .app_data(shared_data.clone())
        .configure(general_routes)
        .configure(course_routes) #1
};
```

We've completed the route configuration. But the code won't compile yet. Let's write the handler function to post a new course.

Step 2: Write the handler function

Recall that an Actix handler function processes an incoming HTTP Request using the data payload and URL parameters sent with the request, and sends back an HTTP response. Let's write the handler for processing a POST request for a new course. Once the new course is created by the handler, it is stored as part of the *AppState* struct, which is then automatically made available to the other handlers in the application. Add the following code to *tutor-nodb/src/handlers.rs*.

Listing 3.7. Handler function for posting a new course

```
// previous imports not shown here
use super::models::Course;
use chrono::Utc;

pub async fn new_course(                                     #1
    new_course: web::Json<Course>,
    app_state: web::Data<AppState>,
) -> HttpResponse {
    println!("Received new course");
    let course_count_for_user = app_state
        .courses
        .lock()
        .unwrap()                                         #2
        .clone()
        .into_iter()                                       #3
        .filter(|course| course.tutor_id == new_course.tutor_id)
        .count();                                         #5
    let new_course = Course {                           #6
        tutor_id: new_course.tutor_id,
        course_id: Some(course_count_for_user + 1),
        course_name: new_course.course_name.clone(),
        posted_time: Some(Utc::now().naive_utc()),
    };
    app_state.courses.lock().unwrap().push(new_course); #7
    HttpResponse::Ok().json("Added course")           #8
}
```

To recap, this handler function

- gets write access to the course collection stored in the application state (*AppState*)
- extracts data payload from the incoming request,
- generates a new course id by calculating number of existing courses for the tutor, and incrementing by 1
- creates a new course instance and
- adds the new course instance to the course collection in *AppState*.

Let's write the test scripts for this function, which we can use for automated testing.

Step 3: Write automated test scripts

Actix web provides supporting utilities for automated testing, over and above what Rust provides. To write tests for Actix services, we first have to start with the basic Rust testing utilities - placing tests within the tests module and annotating it for the compiler. In addition, Actix provides an annotation `# [actix_rt::test]` for the async test functions, to instruct the Actix runtime to execute these tests.

Let's create a test script for posting a new course. For this, we need to construct course details to be posted, and also initialize the application state . These are annotated with steps <5> and <6> in test script shown here.

Add this code in `tutor-nodb/src/handlers.rs`, towards the end of the source file.

Listing 3.8. Test script for posting a new course

```
#[cfg(test)]                                     #1
mod tests {                                     #2
    use super::*;

    use actix_web::http::StatusCode;
    use std::sync::Mutex;

#[actix_rt::test]                                #4
async fn post_course_test() {
    let course = web::Json(Course {             #5
        tutor_id: 1,
        course_name: "Hello, this is test course".into(),
        course_id: None,
        posted_time: None,
    });
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),
    });
    let resp = new_course(course, app_state).await;
    assert_eq!(resp.status(), StatusCode::OK);
}
```

Run the tests from the `tutor-nodb` folder, with the following command:

```
cargo test
```

You should see the test successfully executed with a message that looks similar to this:

```
running 1 test
test handlers::tests::post_course_test ... ok

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

Step 4: Build the service and test the API

Build and run the server from the *tutor-no-db* folder with :

```
cargo run
```

From a commandline run the following curl command: (or you can use a GUI tool like Postman):

```
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
```

You should see the message "Added course" returned from server.

You've now built the API for posting a new course. Next, let's retrieve all existing courses for a tutor.

3.2.5 Get all courses for a tutor

Here we'll implement the handler function to retrieve all courses for a tutor. We know the drill, there are four steps to follow.

Step 1: Define the route configuration

Since we have the foundation of code established, things should be quicker from now.

Let's add a new route in *src/routes.rs*.

```
pub fn course_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/courses")
            .route("/", web::post().to(new_course))
```

```

        .route("/{tutor_id}", web::get().to(get_courses_for_t
    );
}

```

Step 2: Write the handler function

The handler function

- retrieves courses from *AppState*,
- filters courses corresponding to *tutor_id* requested, and
- returns the list.

The code shown here is to be entered in *src/handlers.rs*

Listing 3.9. Handler function to get all courses for a tutor

```

pub async fn get_courses_for_tutor(
    app_state: web::Data<AppState>,
    params: web::Path<(i32)>,
) -> HttpResponse {
    let tutor_id: i32 = params.0;

    let filtered_courses = app_state
        .courses
        .lock()
        .unwrap()
        .clone()
        .into_iter()
        .filter(|course| course.tutor_id == tutor_id)      #1
        .collect::<Vec<Course>>();

    if filtered_courses.len() > 0 {
        HttpResponse::Ok().json(filtered_courses)          #2
    } else {
        HttpResponse::Ok().json("No courses found for tutor".to_str())
    }
}

```

Step 3: Write automated test scripts

In this test script, we will invoke the handler function *get_courses_for_tutor*. This function takes two arguments - application state and a URL path

parameter (denoting tutor id). For example, if the user types the following in the browser, it means he/she wants to see list of all courses with *tutor_id* = 1

localhost:3000/courses/1

Recall that this maps to the route definition in *src/routes.rs*, also shown here for reference:

```
.route("/{tutor_id}", web::get().to(get_courses_for_tutor))
```

The Actix framework automatically passes the application state and the URL path parameter to the handler function *get_courses_for_tutor*, in normal course of execution. However for testing purposes, we would have to manually simulate the function arguments by constructing an application state object and URL path parameter. You will see these steps annotated with <1> and <2> respectively in the test script shown next.

Enter the following test script within the tests module in *src/handlers.rs*.

Listing 3.10. Test script for retrieving courses for a tutor

```
#[actix_rt::test]
async fn get_all_courses_success() {
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),
    );
    let tutor_id: web::Path<(i32)> = web::Path::from((1));
    let resp = get_courses_for_tutor(app_state, tutor_id).awa
    assert_eq!(resp.status(), StatusCode::OK);
}
```

Step 4: Build the service and test the API

Build and run the server from folder *tutor-nodb* with :

```
cargo run
```

Post a few courses from command line as shown (or use a GUI tool like *Postman*):

```
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
```

From a web browser, type the following in URL box:

localhost:3000/courses/1

You should see the courses displayed as shown next:

```
[{"tutor_id":1,"course_id":1,"course_name":"Hello , my first course"}, {"tutor_id":1,"course_id":2,"course_name":"Data Structures and Algorithms"}, {"tutor_id":1,"course_id":3,"course_name":"Machine Learning"}, {"tutor_id":1,"course_id":4,"course_name":"Computer Networks"}, {"tutor_id":1,"course_id":5,"course_name":"Operating Systems"}, {"tutor_id":1,"course_id":6,"course_name":"Database Systems"}, {"tutor_id":1,"course_id":7,"course_name":"Software Engineering"}, {"tutor_id":1,"course_id":8,"course_name":"Compiler Design"}, {"tutor_id":1,"course_id":9,"course_name":"Artificial Intelligence"}, {"tutor_id":1,"course_id":10,"course_name":"Computer Architecture"}]
```

Try posting more courses and verify results.

Our web service is now capable of retrieving course list for a tutor.

3.2.6 Get details of a single course

In this section, we'll implement the handler function to search and retrieve details for a specific course. Let's again go through the defined 4-step process.

Step 1: Define the route configuration

Add a new route as shown here in *src/routes.rs*.

```
pub fn course_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/courses")
            .route("/", web::post().to(new_course))
            .route("/{tutor_id}", web::get().to(get_courses_for_tutor))
            .route("/{tutor_id}/{course_id}", web::get().to(get_course_details));
    )
}
```

Step 2: Write the handler function

The handler function is similar to the previous API (to get all courses for a tutor), except for the additional step to filter on course id also.

Listing 3.11. Handler function to retrieve details for a single course

```
pub async fn get_course_detail(
    app_state: web::Data<AppState>,
    params: web::Path<(i32, i32)>,
) -> HttpResponse {
    let (tutor_id, course_id) = params.0;
    let selected_course = app_state
        .courses
        .lock()
        .unwrap()
        .clone()
        .into_iter()
        .find(|x| x.tutor_id == tutor_id && x.course_id == Some(c
            .ok_or("Course not found")); #2

    if let Ok(course) = selected_course {
        HttpResponse::Ok().json(course)
    } else {
        HttpResponse::Ok().json("Course not found".to_string())
    }
}
```

Step 3: Write automated test scripts

In this test script, we will invoke the handler function *get_course_detail*. This function takes two arguments - application state and URL path parameters. For example, if the user types the following in the browser, it means the user wants to see details of course with *user id* = 1 (first parameter in URL path) and *course id* = 1 (second parameter in URL path).

```
localhost:3000/courses/1/1
```

Recall that this maps to the route definition in *src/routes.rs*, shown next for reference:

```
.route("/{tutor_id}/{course_id}", web::get().to(get_course_detail))
```

The Actix framework automatically passes the application state and the URL path parameters to the handler function *get_course_detail* in normal course of execution. But for testing purposes, we would have to manually simulate the function arguments by constructing an application state object and URL path

parameters. You will see these steps annotated with <1> and <2> respectively in the test script shown.

Add the following test function to *tests* module within *src/handlers.rs*.

Listing 3.12. Test case to retrieve course detail

```
#[actix_rt::test]
async fn get_one_course_success() {
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),
    );
    let params: web::Path<(i32, i32)> = web::Path::from((1, 1
    let resp = get_course_detail(app_state, params).await;
    assert_eq!(resp.status(), StatusCode::OK);
}
```

Step 4: Build the server and test the API

Build and run the server from folder *tutor-nodb* with :

```
cargo run
```

Post two new courses from command line with:

```
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
```

From a web browser type the following in URL box:

```
localhost:3000/courses/1/1
```

You should see the course detail displayed for *tutor_id* = 1 and *course_id* = 1, as shown here:

```
{"tutor_id":1,"course_id":1,"course_name":"Hello , my first course"}
```

You can add more courses, and check if the correct detail is displayed for the other course ids.

Our web service is now capable of retrieving details for a single course.

Note that the tests shown in this chapter are only to demonstrate how to write test scripts for various types of APIs with different types of data payload and URL parameters sent from the web client. Real-world tests would be more exhaustive covering various success and failure scenarios.

In this chapter, you've built a set of RESTful APIs for a tutor web application from scratch starting with data models, routes , application state, and request handlers. You also wrote automated test cases using Actix web's inbuilt test execution support for web applications.

Congratulations, you have built your first web service in Rust!

3.3 Summary

- Actix is a modern, light-weight web framework written in Rust. It provides an async HTTP server that offers safe concurrency and high performance.
- The key components of Actix web we used in this chapter are `HttpServer`, `App`, `routes` , `handlers` , `request extractors`, `HttpResponse` and `application state`. These are the core components needed to build RESTful APIs in Rust using Actix.
- A webservice is a combination of one or more APIs, accessible over HTTP, at a particular domain address and port. APIs can be built using different architectural styles. REST is a popular and intuitive architectural style used to build APIs, and aligns well with the HTTP protocol standards.
- Each RESTful API is configured as a route in Actix. A route is a combination of a *path* that identifies a resource, *HTTP method* and *handler* function.
- A RESTful API call sent from a web or mobile client is received over HTTP by the Actix `HttpServer` listening on a specific port. The request is passed on to the Actix web application registered with it. One or more routes are registered with the Actix web application, which routes the incoming request to a *handler* function (based on *request path* and *HTTP method*).

- Actix provides two types of concurrency - multi-threading and Async I/O. This enables development of high performance web services.
- The Actix HTTP server uses multi-threading concurrency by starting multiple worker threads on startup, equal to the number of logical CPUs in the system. Each thread runs a separate instance of the Actix web application.
- In addition to multi-threading, Actix uses Async I/O, which is another type of concurrency mechanism. This enables an Actix web application to perform other tasks while waiting on I/O on a single thread. Actix has its own Async runtime that is based on *Tokio*, a popular, production-ready async library in Rust.
- Actix allows the web application to define custom application state, and provides a mechanism to safely access this state from each handler function. Since each application instance of Actix runs in a separate thread, Actix provides a safe mechanism to access and mutate this shared state without conflicts or data races.
- At a minimum, a RESTful API implementation in Actix requires a route configuration and a handler function to be added.
- Actix also provides utilities for writing automated test cases.

In the next chapter we will continue with the code built here, and add a persistence layer for the web service, using a relational database.

4 Performing database operations

This chapter covers

- Writing our first async connection to database
- Setting up the web service and writing unit tests
- Creating and querying records from the database

In the previous chapter, we built a web service that uses an *in-memory data store*. In this chapter, we'll enhance that web service. We'll replace the *in-memory data store* with a *relational database*.

Our enhanced web service will expose the same set of APIs as before, but we will now have a proper database to persist the data to disk, because we do not want our data to get lost everytime we restart the web service. As there are many parts to take in, this database-backed web service will be developed iteratively and incrementally over three iterations of code.

In the *first iteration*, we'll learn how to connect asynchronously to a *postgres* database, using a database connection pool, from a vanilla Rust program.

In the *second iteration*, we'll set up the project structure for the Actix-based web service and write unit tests.

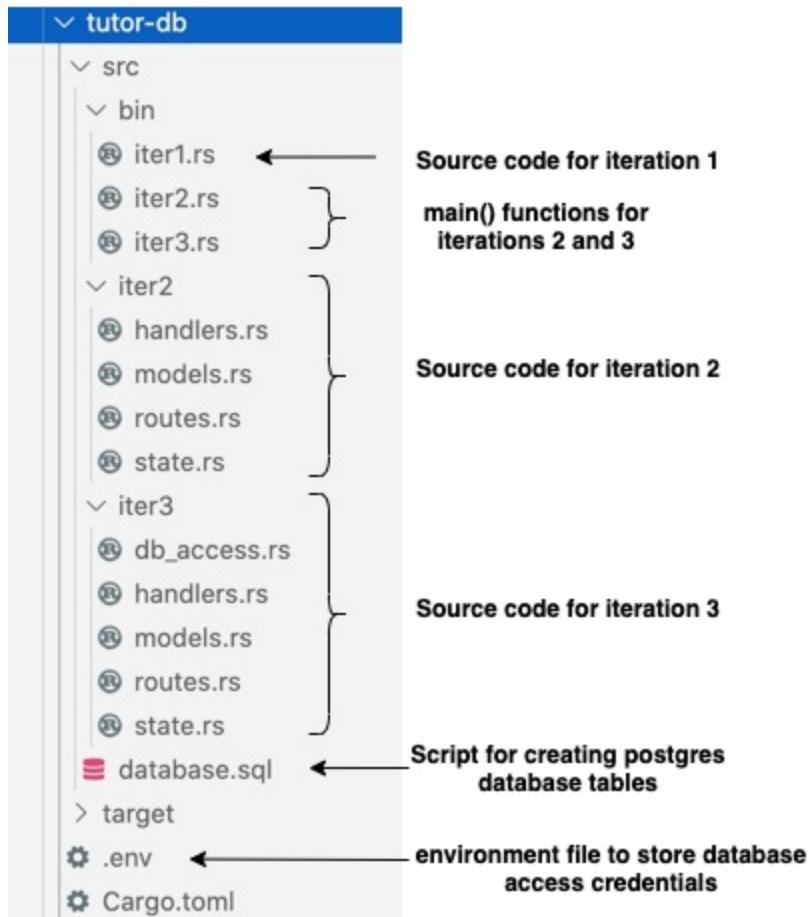
In the *third iteration*, we'll write the actual handler functions to create database records and query the results.

At the end of each iteration, you will have a working version of code that can be inspected, run and tested independently.

The final code structure for this chapter is shown in figure 4.1.

Figure 4.1. Project structure

Project Structure - tutordb



With these goals in mind, let's get started.

Go to the root of the `ezytutors` workspace root (which we created in the previous chapter), and execute the following two steps:

- Add the following to `Cargo.toml`. Note that `tutor-nodb` was the project we created in the previous chapter.

```
[workspace]
members = ["tutor-nodb", "tutor-db"]
```

- Create a new cargo project - `tutor-db`:

```
cargo new tutor-db
cd tutor-db
```

Note that all subsequent command-line statements in this chapter will need to be run from this project root folder (*ezytutors/tutor-db*). To make it easier let's set an environment variable for project root:

```
export PROJECT_ROOT=.
```

Note: The dot at the end of the export statement represents the current directory. Alternatively, replace it with a suitable fully qualified path name.

Environment Variables

In this chapter we will use the following environment variables. Please ensure to set it either manually in your shell session or add it to your shell profile script (e.g. *.bash_profile*).

PROJECT_ROOT: Represents the home directory of the project. For this chapter it is the *tutor-db* root directory, which also contains the *Cargo.toml* file for the project.

DATABASE_USER: Represents the database username that has access (read/write) rights to the database (which we will create later in this chapter).

The complete code for this chapter can be found at
<https://github.com/peshwar9/rust-servers-services-apps/tree/master/chapter4/>.

Software versions

This chapter has been tested with the following versions of software:

- rustc: 1.59.0
- actix-web: 4.2.1
- actix-rt: 2.7.0
- sqlx: 0.6.2
- Platform: Ubuntu 22.04 (LTS) x64

If you have any difficulty in compiling or building the program, you can adjust your development environment to develop and test with these versions.

4.1 Writing our first async connection to database *(Iteration 1)*

In this section, we'll write a simple Rust program to connect to the *postgres* database, and query the database. All code in this section will reside in just one file: *tutor-db/src/bin/iter1.rs*.

4.1.1 Selecting the database and connection library

In this chapter, we'll be using *PostgreSQL* (we will refer to it as simply *postgres* henceforth) as the relational database. *Postgres* is a popular open source relational database that is known for its scalability, reliability, feature set and ability to handle large complicated data workloads.

To connect to *postgres*, we'll use the Rust *sqlx* crate. This crate requires writing queries as raw SQL statements. *sqlx* performs compile-time checking of the query, provides a built-in connection pool, and returns an asynchronous connection to *postgres*. Compile-time checking is very useful to detect and prevent run-time errors.

Having an asynchronous connection to the database for our web service means that our *tutor* web service is free to perform other tasks while waiting on a response from the database. If we were to use a synchronous (hence blocking) connection to the database (such as with *Diesel* ORM), the web service would have to wait until the database operation is completed.

Why use *sqlx*?

Using asynchronous database connections can improve transaction throughput and performance response time of the web service under heavy loads, all other things being equal. Hence the use of *sqlx*.

The primary alternative to *sqlx* is to use *Diesel*, a pure-Rust ORM (object-relational mapper) solution. For those who are used to ORMs from other programming languages and web frameworks, *Diesel* may be a preferred option. But at the time of writing this chapter, *Diesel* does not yet support asynchronous connections to databases. Given that the Actix framework is

asynchronous, it makes the programming model simpler by using async connections to the database too, using a library such as *sqlx*.

Let's start with setting up the database first.

4.1.2 Setting up the database and connecting with async pool

In this section, we'll perform the prerequisites needed to get started with databases. Here are the steps:

1. Add *sqlx* dependency to *Cargo.toml*
2. Install *postgres* and verify installation
3. Create a new database and set up access credentials
4. Define the database model in Rust and create a table in the database
5. Write Rust code to connect to the database and perform a query

For step 5, we'll not use the Actix web server, but instead write a vanilla Rust program. The primary goal of this section is to eliminate database setup and configuration issues, learn to use *sqlx* to connect to databases, and to do a sanity test for database connectivity. By the end of this section, you will have learned to query the *postgres* database using *sqlx* and display query results on your terminal.

Let's look at each step in detail.

Step 1: Add *sqlx* dependencies to *Cargo.toml*

As discussed earlier, we'll use the *sqlx* *async client* to communicate with a *postgres* database. Add the following dependencies in *Cargo.toml* of *tutor-db* project (located in `$PROJECT_ROOT`):

```
[dependencies]
#Actix web framework and run-time #1
actix-web = "4.1.0"
actix-rt = "2.7.0"
#Environment variable access libraries
dotenv = "0.15.0" #2

#Postgres access library
sqlx = {version = "0.6.2", default_features = false, features = [
```

```
# Data serialization library
serde = { version = "1.0.144", features = ["derive"] }          #4

# Other utils
chrono = {version = "0.4.22", features = ["serde"]}           #5

# Openssl for build (if openssl is not already installed on the d
openssl = { version = "0.10.41", features = ["vendored"] }       #6
```

Step 2: Install postgres and verify installation

Please refer to Appendix A: Postgres installation.

Step 3: Create a new database and access credentials

Switch over to the *postgres* account on your development machine/server. If you are on linux, you can use the following command:

```
sudo -i -u postgres
```

You can now access a Postgres prompt (shell) with the following command:

```
psql
```

This will log you into the PostgreSQL prompt, and you can interact with the *postgres* database. You should now be able to see the following prompt (let's call it *psql shell prompt*).

```
postgres=#
```

Now that we are at the *psql shell prompt*, we can create a new database, a new user and associate the user with the database.

First, let's create a database - *ezytutors* with the following command:

```
postgres=# create database ezytutors;
```

Next, create a new user *truuser* with password 'mypassword' (replace username and password with your own) as shown:

```
postgres=# create user truuser with password 'mypassword';
```

Grant access privileges for the newly created user to the *ezytutors* database:

```
postgres=# grant all privileges on database ezytutors to truuser;
```

You can quit the postgres shell prompt with:

```
postgres=# \q
```

Exit out of the postgres user account with:

```
exit
```

You should now be in the prompt of the original user with which you logged into the Linux server (or development machine).

Now ensure you are able to log into the *postgres* database using the new user and password.

Let's first set an environment variable for the database user as follows:

```
export DATABASE_USER=truuser
```

Note: Feel free to replace value of `DATABASE_USER` with the user name you created in the previous step.

On the commandline, you can use the following command to login to the database *ezytutors_* with the database user name created. The **--password** flag is to prompt for password entry:

```
psql -U $DATABASE_USER -d ezytutors --password
```

Type the password at the prompt, and you should get logged into a *psql shell* with the following prompt:

```
ezytutors=>
```

At this prompt type the following to list the databases:

```
\list
```

You'll see the *ezytutors* database listed similar to this:

List of databases						
Name	Owner	Encoding	Collate	Ctype	Access pr	
ezytutors	postgres	UTF8	C.UTF-8	C.UTF-8	=Tc/postgre postgres=CT	
						trouser=CTc

If you're reached this far, great! If not, consult postgres installation and setup instructions for your target development environment at:

<https://www.postgresql.org/docs/12/app-psql.html>

Note: You can also perform the above steps from a GUI admin interface should you choose to install a GUI tool such as cpanel (from a cloud provider) or pgadmin tool (which is available to download for free).

Step 4: Define the Rust database model and create a table

We're now ready to define our database model in the Rust program, and create the database table. There are a couple of ways in which you can do this:

- By using plain database *sql scripts* which are independent of a database access library such as *sqlx*.
- Using *sqlx* CLI

We will use the first approach for this chapter because *sqlx* CLI is in early beta at the time of this writing. But depending on when you are reading this, you may choose to use SQL CLI if there is a stable release by then.

Create a file *database.sql* under *src* folder of project root, and enter the following script:

```
/* Drop table if it already exists*/
drop table if exists ezy_course_c4;
/* Create a table. */
/* Note: Don't put a comma after last field */
create table ezy_course_c4
(
    course_id serial primary key,
    tutor_id INT not null,
    course_name varchar(140) not null,
    posted_time TIMESTAMP default now()
```

```

);
/* Load seed data for testing */
insert into ezy_course_c4
  (course_id,tutor_id, course_name,posted_time)
values(1, 1, 'First course', '2020-12-17 05:40:00');
insert into ezy_course_c4
  (course_id, tutor_id, course_name,posted_time)
values(2, 1, 'Second course', '2020-12-18 05:45:00');

```

We are creating a table with the name *ezy_course_c4*. The c4 suffix is to indicate this is from *chapter 4*, as this allows us to evolve the table definition in a future chapter.

Run the script with the following command from your terminal command prompt. Enter a password if prompted.

```
psql -U $DATABASE_USER -d ezytutors < $PROJECT_ROOT/src/database.
```

This script creates a table called *ezy_course_c4* within the *ezytutors*_database, and loads seed data for testing.

From the SQL shell or admin GUI, run the following sql statement and verify that the records are displayed from database *ezytutors*, table *ezy_course_c4*.

```
psql -U $DATABASE_USER -d ezytutors --password
select * from ezy_course_c4;
```

You should see a result displayed similar to this:

course_id	tutor_id	course_name	posted_time
1	1	First course	2020-12-17 05:40:00
2	1	Second course	2020-12-18 05:45:00

(2 rows)

Step 5: Write code to connect to the database and to query the table

We're now ready to write Rust code to connect to the database! In *src/bin/iter1.rs* under project root, add the following code.

```
use dotenv::dotenv;
use std::env;
```

```

use std::io;
use sqlx::postgres::PgPool;
use chrono::NaiveDateTime;
#[derive(Debug)]                                     #1
pub struct Course {
    pub course_id: i32,
    pub tutor_id: i32,
    pub course_name: String,
    pub posted_time: Option<NaiveDateTime>,
}
#[actix_rt::main]                                     #2
async fn main() -> io::Result<()> {
    dotenv().ok();                                    #3
    let database_url = env::var("DATABASE_URL").expect("DATABASE_UR
    let db_pool = PgPool::connect(&database_url).await.unwrap();
    let course_rows = sqlx::query!(
        r#"select course_id, tutor_id, course_name, posted_time fr
        1
    )
    .fetch_all(&db_pool)
    .await
    .unwrap();
    let mut courses_list = vec![];
    for course_row in course_rows {
        courses_list.push(Course {
            course_id: course_row.course_id,
            tutor_id: course_row.tutor_id,
            course_name: course_row.course_name,
            posted_time: Some(chrono::NaiveDateTime::from(course_r
        })
    }
    println!("Courses = {:?}", courses_list);
    Ok(())
}

```

Create a `.env` file in the project root directory and make the following entry:

```
DATABASE_URL=postgres://<my-user>:<mypassword>@127.0.0.1:5432/ezy
```

Replace `<my-user>` and `<mypassword>` with the userid and password that you used while setting up the database. 5432 refers to the default port where the postgresql server runs and `ezytutors` is the name of the database we wish to connect to.

Run the code with the following command:

```
cargo run --bin iter1
```

Note that by using the `--bin` flag, we are telling the *Cargo* tool to run the `main()` function located in `iter1.rs` from the `$PROJECT_ROOT/src/bin` directory.

You should see the list of query results displayed to your terminal as shown here.

```
Courses = [Course { course_id: 1, tutor_id: 1, course_name: "First Course" }]
```

Great! We are now able to connect to the database from a Rust program using `sqlx` crate.

Running the program from workspace root instead of project root

Note that you can also choose to run the program from the *workspace root* (`ezytutors` directory) instead of the *project root* (`tutordb` directory). If so, you need to add an additional flag to the cargo run command as shown:

```
cargo run --bin iter1 -p tutordb
```

Since the `ezytutors` workspace contains many projects, we need to tell the *cargo* tool which project to execute. This is done by using the `-p` flag along with the project name (`tutordb`).

Note also that if you choose to do this, the `.env` file containing the database access credentials should be located within the *workspace root* as opposed to the *project root*. But in this chapter, we will follow the convention of executing the program from the project root only.

4.2 Setting up the web service and writing unit tests (Iteration 2)

Now that we know how to connect to a *postgres* database using `sqlx`, let's get back to writing our database-backed web service. By the end of this section, you will have a code structure for the web service that includes routes, database model, application state, `main()` function, unit test scripts for the

three APIs, and skeletal code for the handler functions. This section serves as an interim checkpoint. You will be able to compile the web service, and ensure there are no compilation errors, before proceeding any further. But the web service won't perform anything useful until we write the handler functions in the next section. Here are the steps we'll perform in this section:

1. Setup dependencies & routes
2. Setup the application state and the data model
3. Setup the connection pool using dependency injection
4. Write unit tests

4.2.1 Setup dependencies and routes

Create a folder called *iter2* under *\$PROJECT_ROOT/src*. The code for this section will be organized as follows:

- *src/bin/iter2.rs* : contains the *main()* function
- *src/iter2/routes.rs*: contains *routes*
- *src/iter2/handlers.rs*: contains *handler* functions
- *src/iter2/models.rs*: contains the data structure to represent a *course* and utility methods
- *src/iter2/state.rs*: Application state containing the dependencies injected into each thread of application execution

Basically, the *main()* function will be in the *iter.rs* file under *src/bin* folder of project root, and the rest of the files will be placed under *src/iter2* folder.

We'll reuse the same set of routes defined in the previous chapter. The code to be placed in *\$PROJECT_ROOT/src/iter2/routes.rs* is shown here:

Listing 4.1. Routes for Tutor web service

```
use super::handlers::*;
use actix_web::web;

pub fn general_routes(cfg: &mut web::ServiceConfig) {
    cfg.route("/health", web::get().to(health_check_handler));
}
```

```

pub fn course_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/courses")
            .route("/", web::post().to(post_new_course))           #1
            .route("/{tutor_id}", web::get().to(get_courses_for_tutor))
            .route("/{tutor_id}/{course_id}", web::get().to(get_course_))
    );
}

```

4.2.2 Setup the application state and the data model

Let's define the data model in `src/iter2/models.rs` under project root.

Here we'll define a data structure to represent a course. We'll also write a utility method that accepts the JSON data payload sent with the HTTP POST request, and converts it into the Rust `Course` data structure. Place the following code in `$PROJECT_ROOT/src/iter2/models.rs`.

Listing 4.2. Data model for the tutor web service

```

use actix_web::web;
use chrono::NaiveDateTime;
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize, Debug, Clone)]           #1
pub struct Course {                                     #1
    pub course_id: i32,
    pub tutor_id: i32,
    pub course_name: String,
    pub posted_time: Option<NaiveDateTime>,
}
impl From<web::Json<Course>> for Course {          #2
    fn from(course: web::Json<Course>) -> Self {
        Course {
            course_id: course.course_id,
            tutor_id: course.tutor_id,
            course_name: course.course_name.clone(),
            posted_time: course.posted_time,
        }
    }
}

```

For connecting to `postgres`, we'll have to define a database connection pool,

and make it available across worker threads. We can achieve this by defining a connection pool as part of the application state.

Add the following code to `$PROJECT_ROOT/src/iter2/state.rs`:

```
use sqlx::postgres::PgPool;
use std::sync::Mutex;
pub struct AppState {
    pub health_check_response: String,
    pub visit_count: Mutex<u32>,
    pub db: PgPool,
}
```

In the `AppState` struct, we have retained the two fields from the previous chapter needed for health check response, and added an additional field `db` which represents the *sqlx postgres connection pool*.

With the application state definition done, it's time to write the `main()` function for the web service.

4.2.3 Setup connection pool using dependency injection

In the `main()` function for the web service, we will perform the following:

- Retrieve environment variable `DATABASE_URL` for credentials to connect to the database
- Create a `sqlx` connection pool
- Create application state and add connection pool to it
- Create a new *Actix web application* and configure it with routes. Inject `AppState` struct as a dependency into the web application so it is made available to handler functions across threads
- Initialize *Actix web server* with the *web application* and run the server

The code listing for the `main()` function in `$PROJECT_ROOT/src/bin/iter2.rs` is shown here.

Listing 4.3. Tutor web service main() function

```
use actix_web::{web, App, HttpServer};
use dotenv::dotenv;
```

```

use sqlx::postgres::PgPool;
use std::env;
use std::io;
use std::sync::Mutex;

#[path = "../iter2/handlers.rs"]
mod handlers;
#[path = "../iter2/models.rs"]
mod models;
#[path = "../iter2/routes.rs"]
mod routes;
#[path = "../iter2/state.rs"]
mod state;

use routes::*;
use state::AppState;

#[actix_rt::main]
async fn main() -> io::Result<()> {
    dotenv().ok();                                #1

    let database_url = env::var("DATABASE_URL").expect("DATABASE_U
    let db_pool = PgPool::connect(&database_url).await.unwrap();
    // Construct App State
    let shared_data = web::Data::new(AppState {
        health_check_response: "I'm good. You've already asked me
        visit_count: Mutex::new(0),
        db: db_pool,
    });
    //Construct app and configure routes
    let app = move || {
        App::new()
            .app_data(shared_data.clone())           #3
            .configure(general_routes)
            .configure(course_routes)
    };
    //Start HTTP server

    HttpServer::new(app).bind("127.0.0.1:3000")?.run().await
}

```

Rest of the `main()` function is similar to what we wrote in the previous chapter. Let's also write the handler functions in `$PROJECT_ROOT/src/iter2/handlers.rs`.

Listing 4.4. Handler functions skeleton

```
use super::models::Course;
use super::state::AppState;
use actix_web::{web, HttpResponse};

pub async fn health_check_handler(app_state: web::Data<AppState>)
    let health_check_response = &app_state.health_check_response;
    let mut visit_count = app_state.visit_count.lock().unwrap();
    let response = format!("{} {} times", health_check_response, v
        *visit_count += 1;
        HttpResponse::Ok().json(&response)
    } #1

pub async fn get_courses_for_tutor(
    _app_state: web::Data<AppState>,
    _params: web::Path<(i32, )>,
) -> HttpResponse {
    HttpResponse::Ok().json("success")
}

pub async fn get_course_details(
    _app_state: web::Data<AppState>,
    _params: web::Path<(i32, i32)>,
) -> HttpResponse {
    HttpResponse::Ok().json("success")
}

pub async fn post_new_course(
    _new_course: web::Json<Course>,
    _app_state: web::Data<AppState>,
) -> HttpResponse {
    HttpResponse::Ok().json("success")
}
```

We've written the skeletal code for the three *tutor* handler functions. These don't do much except to return a success response for now. The goal is to verify that the code for web service compiles without errors before we implement the database access logic in the next section.

Verify the code with the following command from the *project root*:

```
cargo check --bin iter2
```

The code should compile without errors, and the server should start up. You

may see a few warnings related to unused variables, but let's ignore it for now as this is only an interim checkpoint. Let's now write the unit tests for the three handler functions.

4.2.4 Write the unit tests

In the previous section, we wrote dummy handler functions that simply return a success response. In this section, let's write the unit tests that invoke these handler functions. In the process, you'll learn how to simulate HTTP request parameters (which would otherwise come through an external API call), simulate Application state being passed from the Actix framework to the handler function, and how to check for responses from the handler functions in the test functions.

We'll write three unit test functions to test the three corresponding handler functions we wrote in the previous section i.e. to get all courses for a tutor, to get course details for an individual course, and to post a new course.

Let's now add the following unit test code to `$PROJECT_ROOT/src/iter2/handlers.rs` file.

Listing 4.5. Unit tests for the handler functions

```
#[cfg(test)]
mod tests {
    use super::*;

    #[actix_rt::test]
    async fn get_all_courses_success() {
        dotenv().ok();
        let database_url = env::var("DATABASE_URL").expect("DATABASE_URL");
        let pool: PgPool = PgPool::connect(&database_url).await.unwrap();
        let app_state: web::Data<AppState> = web::Data::new(AppState {
            health_check_response: "".to_string(),
            visit_count: Mutex::new(0),
        });
        let client = Client::new();
        let resp = client.get("/api/courses").send().await.unwrap();
        assert_eq!(resp.status(), StatusCode::OK);
        let body: String = resp.text().await.unwrap();
        assert_eq!(body, "[{");
    }
}
```

```

        db: pool,
    });
    let tutor_id: web::Path<(i32,)> = web::Path::from((1,));
    let resp = get_courses_for_tutor(app_state, tutor_id).await;
    assert_eq!(resp.status(), StatusCode::OK);
}

#[actix_rt::test]
async fn get_course_detail_test() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATABASE_URL");
    let pool: PgPool = PgPool::connect(&database_url).await.unwrap();
    let app_state: ... #8
    let params: web::Path<(i32, i32)> = web::Path::from((1, 2));
    let resp = get_course_details(app_state, params).await;
    assert_eq!(resp.status(), StatusCode::OK);
}

#[actix_rt::test]
async fn post_course_success() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATABASE_URL");
    let pool: PgPool = PgPool::connect(&database_url).await.unwrap();
    let app_state: ... #8
    let new_course_msg = Course {
        course_id: 1,
        tutor_id: 1,
        course_name: "This is the next course".into(),
        posted_time: Some(NaiveDate::from_ymd(2020, 9, 17).and_hms(0, 0, 0)),
    };
    let course_param = web::Json(new_course_msg);
    let resp = post_new_course(course_param, app_state).await;
    assert_eq!(resp.status(), StatusCode::OK);
}
}
}

```

The code is annotated for one of the test functions. The same concepts hold good for the other two test functions as well, and you should be able to read the test function code and follow it without much difficulty.

Let's run the unit tests with:

```
cargo test --bin iter2
```

You should see the three tests pass successfully with the following message:

```
running 3 tests
test handlers::tests::get_all_courses_success ... ok
test handlers::tests::post_course_success ... ok
test handlers::tests::get_course_detail_test ... ok

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

The tests pass even though we haven't written any database access logic because we are returning an unconditional success response from the handlers. We'll fix that in the next section. But we have built the basic project structure with all the required pieces (routes, application state, main() function, handlers and unit tests) and now know how to tie all of them together.

4.3 Creating and querying records from the database (*Iteration 3*)

In this section, we'll write the database access code for the tutor APIs.

Create a folder named *iter3* under *\$PROJECT_ROOT/src*. The code for this section will be organized as follows:

- *src/bin/iter3.rs* : contains the *main()* function
- *src/iter3/routes.rs*: contains *routes*
- *src/iter3/handlers.rs*: contains *handler* functions
- *src/iter3/models.rs*: contains the data structure to represent a *course* and a few utility methods
- *src/iter3/state.rs*: Application state containing the dependencies injected into each thread of application execution
- *src/iter3/db_access.rs*: We don't want the database access logic to be a part of the handler function, to adhere to the *single responsibility principle*. So, we'll create a new file *\$PROJECT_ROOT/src/iter3/db_access.rs* for the database access logic. Separating out database access will also be helpful if we want to switch databases (say from postgres to Mysql) in future, in which case we can just rewrite the database access functions with the new database while retaining the same handler functions and database access function signatures.

Of the files listed for this iteration, we can reuse the code for *routes.rs*, *state.rs* and *models.rs* from *iteration 2*. That leaves us to focus our efforts in this section primarily on making the required adjustments to the *main()* function and handler code, and to write the core database access logic.

Let's look at the code for database access in three parts, each part corresponding to one of the APIs.

4.3.1 Writing database access functions

The steps for using *sqlx* to query records from *postgres* tables are listed here:

- Construct the SQL query using *sql query!* macro
- Execute the query using *fetch_all()* method passing the connection pool
- Extract the results and convert them into a Rust struct that can be returned from the function.

The code in *\$PROJECT_ROOT/src/iter3/db_access.rs* is shown here.

Listing 4.6. Database access code for retrieving all courses for a tutor

```
use super::models::Course;
use sqlx::postgres::PgPool;

pub async fn get_courses_for_tutor_db(pool: &PgPool, tutor_id: i3
    // Prepare SQL statement
    let course_rows = sqlx::query!(
        "SELECT tutor_id, course_id, course_name, posted_time FROM
        tutor_id
    )
    .fetch_all(pool)                                #2
    .await
    .unwrap();
    // Extract result
    course_rows                                     #3
        .iter()
        .map(|course_row| Course {
            course_id: course_row.course_id,
            tutor_id: course_row.tutor_id,
            course_name: course_row.course_name.clone(),
            posted_time: Some(chrono::NaiveDateTime::from(course_r
        })
```

```
    .collect()
}
```

We're using the *fetch_all()* method to retrieve all records from the database that match the sql query. The *fetch_all()* method accepts a postgres connection pool as a parameter. The *await* keyword after *fetch_all()* denotes that we are making an asynchronous call to the postgres database using the *sqlx* crate.

Note the use of *iter()* method to convert the retrieved database records into a Rust *iterator*. The *map()* function then converts each database row (returned by the *iterator*) into a Rust data structure of type *Course*.

Finally, the results from applying the *map()* function on all database records are accumulated into a Rust *Vec* data type by using the *collect()* method. The vector of *Course* struct instances is then returned from the function.

Note also the use of *chrono* module to convert the *posted_time* value of a course retrieved from the database, into a *NaiveDateTime* type from the *chrono* crate.

Overall, you'll notice that the code is quite concise due to the use of elegant functional programming constructs that Rust provides.

The code for retrieving the course details given a *course-id* and *tutor-id* is largely similar. The main difference is the use of *fetch_one()* method instead of *fetch_all()* that we used previously, as here we are retrieving details for a single course. Place this code in the same file i.e.
\$PROJECT_ROOT/src/iter3/db_access.rs.

Listing 4.7. Database access code for retrieving details of a single course

```
pub async fn get_course_details_db(pool: &PgPool, tutor_id: i32,
    // Prepare SQL statement
    let course_row = sqlx::query!(
        "SELECT tutor_id, course_id, course_name, posted_time FROM
        tutor_id, course_id
    )
    .fetch_one(pool)                                #2
    .await
```

```

.unwrap();
// Execute query
Course {
    course_id: course_row.course_id,
    tutor_id: course_row.tutor_id,
    course_name: course_row.course_name.clone(),
    posted_time: Some(chrono::NaiveDateTime::from(course_row.p
}
}

```

Lastly, we'll look at the database access code to post a new course. The query is constructed, and then executed. The inserted course is then retrieved, converted into a Rust struct and returned from the function. Place the following code in `$PROJECT_ROOT/src/iter3/db_access.rs`.

Listing 4.8. Database access code for posting a new course

```

pub async fn post_new_course_db(pool: &PgPool, new_course: Course

let course_row = sqlx::query!("insert into ezy_course_c4 (cour
    .fetch_one(pool)                                            #
    .await.unwrap();
//Retrieve result
Course {                                                       #
    course_id: course_row.course_id,
    tutor_id: course_row.tutor_id,
    course_name: course_row.course_name.clone(),
    posted_time: Some(chrono::NaiveDateTime::from(course_row.p
}
}

```

Note that we're not passing the `posted_time` value to the `insert` query. This is because, while creating the table in the database, we have set the default value of this field to the system generated current time. Refer to the file `$PROJECT_ROOT/src/database.sql` where this default is defined as shown:

```
posted_time TIMESTAMP default now()
```

Note on using MySQL instead of Postgres database

Note that the `sql` crate supports both MySQL and SQLite, in addition to `Postgres`. Readers who prefer to follow along this chapter using a MySQL database in place of Postgres can refer to the instructions for the `sqlx` crate

repository at <https://github.com/launchbadge/sqlx>.

However, one thing to note is that the *SQL* syntax supported for MySQL differs from that of Postgres, so the query statements listed in this chapter need some modifications to use with MySQL. For example, while using MySQL, the \$ sign used to denote parameters (eg \$1) should be replaced with a question mark (?). Also, Postgres supports a *returning* clause in SQL statement that can be used to return values of columns modified by an insert, update or delete operation, but MySQL does not support the *returning* clause directly.

This completes the code for database access. Next, let's look at the *handler* functions that invoke these database access functions.

4.3.2 Writing handler functions

We've so far seen the code for database access. We now need to invoke these database functions from the corresponding handler functions. Recall that the handler functions are invoked by the Actix framework based on the API route (defined in *routes.rs*) on which the HTTP request arrives (eg POST new course, GET courses for tutor etc).

The code for the handler functions to be placed in *\$PROJECT_ROOT/src/iter3/handlers.rs* is shown here.

Listing 4.9. Handler function for retrieving query results

```
use super::db_access::*;
use super::models::Course;
use super::state::AppState;
use std::convert::TryFrom;

use actix_web::{web, HttpResponse};

pub async fn health_check_handler(app_state: web::Data<AppState>)
{
    let health_check_response = &app_state.health_check_response;
    let mut visit_count = app_state.visit_count.lock().unwrap();
    let response = format!("{} {} times", health_check_response, v
        *visit_count += 1;
    HttpResponse::Ok().json(&response)
}
```

```

}

pub async fn get_courses_for_tutor(
    app_state: web::Data<AppState>,
    params: web::Path<(i32,)>,
) -> HttpResponse {
    let tuple = params.0;                      #1
    let tutor_id: i32 = i32::try_from(tuple.0).unwrap();    #2
    let courses = get_courses_for_tutor_db(&app_state.db, tutor_id)
    HttpResponse::Ok().json(courses)
}

pub async fn get_course_details(
    app_state: web::Data<AppState>,
    params: web::Path<(i32, i32)>,
) -> HttpResponse {
    let tuple = params;
    let tutor_id: i32 = i32::try_from(tuple.0).unwrap();
    let course_id: i32 = i32::try_from(tuple.1).unwrap();    #5
    let course = get_course_details_db(&app_state.db, tutor_id, co
    HttpResponse::Ok().json(course)
}

pub async fn post_new_course(
    new_course: web::Json<Course>,
    app_state: web::Data<AppState>,
) -> HttpResponse {
    let course = post_new_course_db(&app_state.db, new_course.into
    HttpResponse::Ok().json(course)
}

```

In the code listing shown earlier, each of the handler functions is fairly straightforward and performs steps similar to those listed here:

1. Extract connection pool from Application state (*appstate.db*)
2. Extract parameters sent as part of the HTTP Request (*params* argument)
3. Invoke the corresponding database access function (the function names suffixed with *db*)
4. Return the result from the database access function as an HTTP Response

Let's understand these steps with the example of the handler function *get_course_details()*. This function is called whenever there is an HTTP

request that arrives on the route "/{tutor_id}/{course_id}". Example of such a request is <http://localhost:3000/courses/1/2> where the http client (e.g. internet browser) is requesting to see details of a course which has a tutor-id of 1 and course-id of 2.

Let's go through the code in detail for this handler function, in slow-mode:

In order to extract the course details for a given *tutor-id* and *course-id*, we need to talk to the database. But the handler function does not know (nor does it need to know, in keeping with the *single responsibility principle* of good software design) how to talk to the database. So it will have to rely on the database access function *get_course_details_db()*, which we wrote in the source file *\$PROJECT_ROOT/src/iter3/db_access.rs*.

This is the signature of this function:

```
pub async fn get_course_details_db(pool: &PgPool, tutor_id: i32,
```

In order to invoke the database access function, the handler function needs to pass three parameters: a *database connection pool*, *tutor-id* and *course-id*.

The connection pool is available as part of the application state object. In the *main()* function of iteration 2, we already saw the code for how the application state is constructed with the connection pool, and then injected into the Actix web application instance. Every Actix handler function will then automatically have access to application state as a parameter (which is automatically populated by the Actix framework when the handler is invoked).

As a result, in this handler, the first parameter *app_state* represents a value of type *AppState* (recall that this struct is defined in *\$PROJECT_ROOT/src/iter3/state.rs*), whose definition is reproduced here:

```
pub struct AppState {  
    pub health_check_response: String,  
    pub visit_count: Mutex<u32>,  
    pub db: PgPool,  
}
```

Hence *app_state.db* refers to the *db* member of struct *AppState*, and

represents the connection pool which can be passed to the database function `get_course_details_db()`.

The next two parameters to pass to the database access function are *tutor-id* and *course-id*. Note that these are available as part of an incoming HTTP request of the form `http(s)://{{domain}}:{{port}}/{{tutor-id}}/{{course-id}}`. In order to extract the parameters from the request, the *Actix web framework* provides utilities called *extractors*. An *extractor* can be accessed as an argument to the handler function (similar to application state we saw earlier). In our case, as we are expecting two numeric parameters from the HTTP request, the handler function signature has a parameter of type `web::Path<(i32, i32)>`, which basically yields a tuple containing two integers of type `(i32, i32)`. In order to extract the value of the *tutor-id* and *course-id* from `params`, we will have to perform a two-step process.

The following line

```
let tuple = params.0;
```

provides a tuple of form `(i32, i32)`.

Then the following two lines are used to extract and convert the *tutor-id* and *course-id* from `i32` to `i32` type (which is the type expected by the database access function):

```
let tutor_id: i32 = i32::try_from(tuple.0).unwrap();
let course_id: i32 = i32::try_from(tuple.1).unwrap();
```

Now we can invoke the database access function with the application state , tutor-id and course-id as shown:

```
let course = get_course_details_db(&app_state.db, tutor_id, co
```

Finally, we take the return value of type *Course* from the database function, serialize it to *Json* type and embed it into an HTTP response with *success status code*, all in a succinct expression (you can now see why Rust rocks!):

```
HttpResponse::Ok().json(course)
```

The other two handler functions are similar in structure to what we've just

seen.

Recall that in the source file *handlers.rs*, we also had the handler function for health check and the unit tests. These remain unchanged from the previous iteration. Note that error handling has been excluded from this iteration, to put the learning focus on database access.

4.3.3 Writing the main() function for the database-backed web service

We've written the database access and handler functions. Let's complete the final piece of code before we can test our web service. Add the following code to the *main()* function in *\$PROJECT_ROOT/src/bin/iter3.rs*.

Listing 4.10. main() function for the third iteration

```
use actix_web::{web, App, HttpServer};
use dotenv::dotenv;
use sqlx::postgres::PgPool;
use std::env;
use std::io;
use std::sync::Mutex;

#[path = "../iter3/db_access.rs"]
mod db_access;
#[path = "../iter3/handlers.rs"]
mod handlers;
#[path = "../iter3/models.rs"]
mod models;
#[path = "../iter3/routes.rs"]
mod routes;
#[path = "../iter3/state.rs"]
mod state;

use routes::*;

use state::AppState;

#[actix_rt::main]
async fn main() -> io::Result<()> {
    dotenv().ok();

    let database_url = env::var("DATABASE_URL").expect("DATABASE_U
    let db_pool = PgPool::connect(&database_url).await.unwrap();
```

```

let shared_data = web::Data::new(AppState {                         #1
    health_check_response: "I'm good. You've already asked me",
    visit_count: Mutex::new(0),
    db: db_pool,
});

let app = move || {                                         #2
    App::new()
        .app_data(shared_data.clone())                      #3
        .configure(general_routes)                         #4
        .configure(course_routes)                          #4
};

//Start HTTP server

HttpServer::new(app).bind("127.0.0.1:3000")?.run().await      #
}

```

We're now ready to test and run the web service. First let's run the automated tests with:

```
cargo test --bin iter3
```

You should see the three test cases execute successfully as shown:

```

running 3 tests
test handlers::tests::post_course_success ... ok
test handlers::tests::get_all_courses_success ... ok
test handlers::tests::get_course_detail_test ... ok

```

Note: if you run the *cargo test* command more than once, the program will exit with an error. This is because we are trying to insert a record with the same *course_id* twice. To get around this, log into the *psql* shell and run the following command:

```
delete from ezy_course_c4 where course_id=3;
```

We are inserting a record with *course_id* value of 3 in the test function. Once we delete this database record, we can rerun the test.

In order to make this step easier, this delete sql statement can be placed within a script file. The file *\$PROJECT_ROOT/iter3-test-clean.sql* contains

this script if you'd like to use it. Execute the script as follows:

```
psql -U $DATABASE_USER -d ezytutors --password < $PROJECT_ROOT/it
```

You can now rerun:

```
cargo test --bin iter3
```

Let's now run the server:

```
cargo run --bin iter3
```

From a browser , enter the following url to retrieve query results for *tutor id* 1.

<http://localhost:3000/courses/1>

Or if you are behind a firewall you can use curl to run it:

```
curl localhost:3000/courses/1
```

You should see something similar to what's shown here as the response:

```
[{"course_id":1,"tutor_id":1,"course_name":"First course","posted
```

You will find three query results in your list. We had added two courses as part of *database.sql* script. We then added a new course using the unit tests.

Let's next test posting a new course using *curl*:

```
curl -X POST localhost:3000/courses/ \
-H "Content-Type: application/json" \
-d '{"tutor_id":1, "course_id":4, "course_name":"Fourth course"}'
```

You should see a response from the Actix web server similar to this:

```
{"course_id":4,"tutor_id":1,"course_name":"Fourth course","posted
```

You can now try to retrieve details for the newly posted course, as shown here, from a browser:

<http://localhost:3000/courses/1/4>

Note: If you are behind a firewall run this command with *curl* as previously suggested.

You'll see a result similar to this, in the browser:

```
{"course_id":4,"tutor_id":1,"course_name":"Fourth course","posted"
```

This concludes *iteration 3*.

With this, we have completed the implementation of three APIs for the *tutor web service* backed by a database store. We have built the functionality to post a new course, persist it to the database, and then query the database for a list of courses and individual course details. Congratulations!

You may have noticed that in this chapter, we covered only the *happy path* scenarios, and did not account for, or handle, any errors that might occur. But this is unrealistic as many things can go wrong in a distributed web application. This will be discussed in the next chapter. In addition, we'll also cover how we can secure our APIs in the next chapter.

4.4 Summary

- *SQLx* is a Rust crate that provides asynchronous database access to many databases including *postgres* and *MySQL*. It has built in connection pooling.
- Connecting to a database from Actix using *sqlx* includes the following three broad steps: 1) In the *main()* function of the web service, create a *sqlx* connection pool and inject it into application state, 2) In the *handler* function, access the connection pool and pass it to the database access function, 3) In the database access function, construct the query and execute it on the connection pool.
- The web service with its three APIs was built in this chapter over three iterations: 1) In *iteration 1*, we configured the database, configured *sqlx* connection to the database, and tested the connection through a vanilla Rust program (not with Actix web server) 2) In *iteration 2*, we setup the database model, routes , state and the *main()* function for the web service. 3) In *iteration 3*, we wrote the database access code for the three

APIs along with the unit tests. The codebase for each of the iterations can be built and tested independently as part of the learning path.

Our tutor web service is now functional, but it does not yet have the ability to handle errors or to authenticate users making the API calls. In the next chapter, we will cover these topics.

5 Handling Errors

This chapter covers

- Setting up the project structure
- Basics of error handling in Rust and Actix Web
- Defining a custom error handler
- Error handling for retrieving all courses
- Error handling for retrieving course details
- Error handling for posting a new course
- Summary

In the previous chapter, we wrote the code to post and retrieve courses through an API. But what we demonstrated and tested were the *happy path* scenarios. In the real world however, many types of failures can occur. The database server may be unavailable, the *tutor id* provided in request may be invalid, there may be a web server error, and so on. It is important that our web service is able to detect the errors , handle them gracefully and send a meaningful error message back to the user or client sending the API request. This is done through *error handling*, which is the focus of this chapter. Error handling is important not just for stability of our web service, but also to provide a good user experience.

Figure 5.1. Unifying error handling in Rust

Unifying error handling for Web services

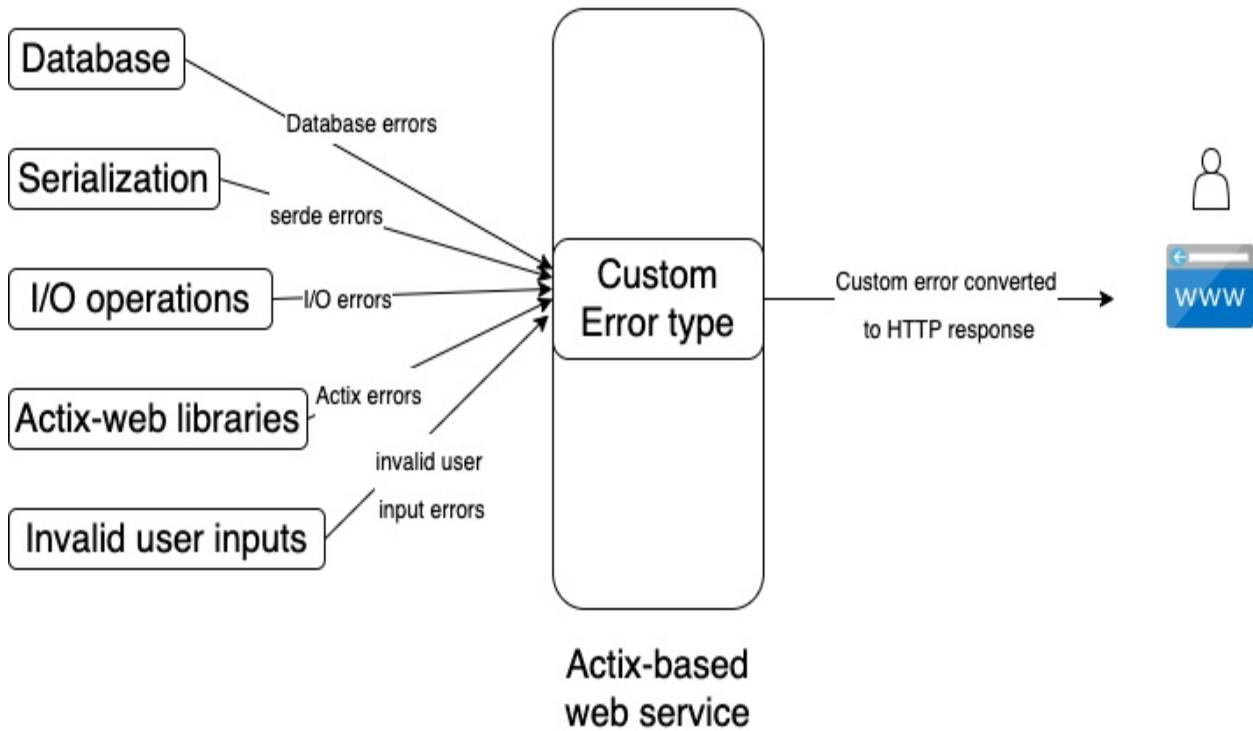


Figure 5.1 summarizes the error handling approach that we will adopt in this chapter. We'll add custom error handling to our web service that unifies different types of errors that can be encountered in the application. The outcome will be that whenever there is an invalid request or unexpected malfunction in server code execution, the client will receive a meaningful and appropriate HTTP status code and error message. To achieve this, we will use a combination of the core Rust features for error handling and the features provided by Actix, while customizing the error handling for our application.

5.1 Setting up the project structure

We will use the code built in the previous chapter as the starting base to add error handling. If you've been following along, you can use your own code from chapter 4 to start adding error handling. Alternatively, clone the following repo: <https://github.com/peshwar9/rust-servers-services-apps> and use the code for *iteration 3* from *chapter 4* as the starting point. We'll build the code in this chapter as *iteration 4*, so first go to the project root

(`ezytutors/tutor-db`) and create a new folder `iter4` under `src`.

The code for this section will be organized as follows:

- `src/bin/iter4.rs` : `main()` function.
- `src/iter4/routes.rs`: Contains routes.
- `src/iter4/handlers.rs`: Handler functions.
- `src/iter4/models.rs`: Data structure to represent a *Course* and utility methods.
- `src/iter4/state.rs`: Application state containing the dependencies that are injected into each thread of application execution.
- `src/iter4/db_access.rs`: Database access code separated out from the handler function, for modularity
- `src/iter4/errors.rs`: Custom error data structure and associated error handling functions

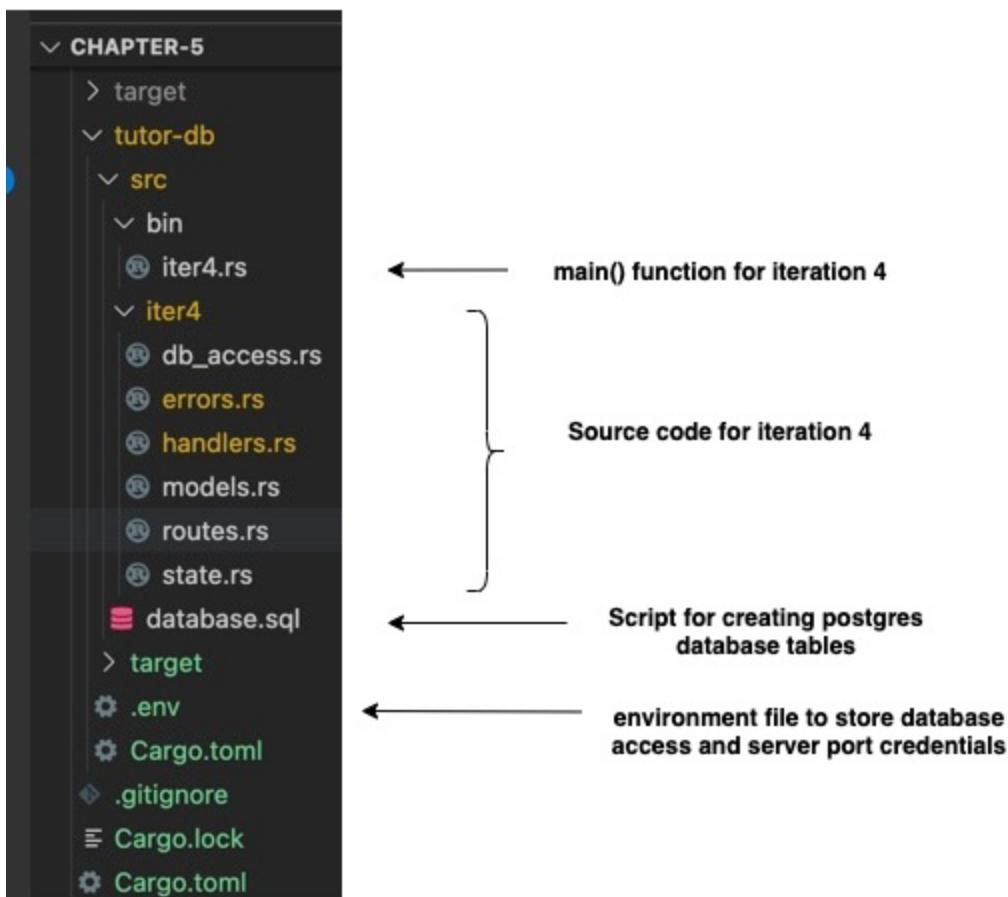
Of the files listed,

- there will be no changes to the source code for `routes.rs`, `models.rs` or `state.rs` compared to chapter 4.
- For `handlers.rs` and `db_access.rs`, we can start with the respective code from chapter 4, but we will modify them to incorporate custom error handling.
- `errors.rs` is a new source file that we'll add.

The project structure should look similar to that shown in figure 5.1

Figure 5.2. Project structure

Project Structure - Error handling



Let's also create a new version of the database tables for this chapter by following these steps:

1. Amend the *database.sql* script from the previous chapter to look like this:

```
/* Drop table if it already exists*/
drop table if exists ezy_course_c5;
/* Create a table. */
/* Note: Don't put a comma after last field */
create table ezy_course_c5
(
    course_id serial primary key,
    tutor_id INT not null,
    course_name varchar(140) not null,
    posted_time TIMESTAMP default now()
);
```

```

/* Load seed data for testing */
insert into ezy_course_c5
    (course_id, tutor_id, course_name, posted_time)
values(1, 1, 'First course', '2021-03-17 05:40:00');
insert into ezy_course_c5
    (course_id, tutor_id, course_name, posted_time)
values(2, 1, 'Second course', '2021-03-18 05:45:00');

```

Note that the main change we have done to the script from the last chapter is to change the name of the table from *ezy_course_c4* to *ezy_course_c5*.

2. Run the script from the command line as shown to create the table and load sample data:

```
psql -U <user-name> -d ezytutors < database.sql
```

Ensure to provide the right path to the *database.sql* file, and enter the password if prompted.

3. After creating the new table, we need to give permission to this new table for the database user. Run the following commands from the terminal command-line.

```

psql -U <user-name> -d ezytutors // Login to psql shell
GRANT ALL PRIVILEGES ON TABLE __ezy_course_c5__ to <user-name>
\q                                // Quit the psql shell

```

Replace the <user-name> with your own, and execute the commands.

4. Write the main() function: From the previous chapter, copy *src/bin/iter3.rs* into your project directory for this chapter under *src/bin/iter4.rs*, and modify references to *iter3* with *iter4*. The final code for *iter4.rs* should look as shown:

```

use actix_web::{web, App, HttpServer};
use dotenv::dotenv;
use sqlx::postgres::PgPool;
use std::env;
use std::io;
use std::sync::Mutex;

#[path = "../iter4/db_access.rs"] #
```

```

mod db_access;
#[path = "../iter4/errors.rs"]
mod errors;
#[path = "../iter4/handlers.rs"]
mod handlers;
#[path = "../iter4/models.rs"]
mod models;
#[path = "../iter4/routes.rs"]
mod routes;
#[path = "../iter4/state.rs"]
mod state;

use routes::*;

use state::AppState;

#[actix_rt::main]
async fn main() -> io::Result<()> {
    dotenv().ok();

    let database_url = env::var("DATABASE_URL").expect("DATABASE_");
    let db_pool = PgPool::connect(&database_url).await.unwrap();
    // Construct App State
    let shared_data = web::Data::new(AppState {
        health_check_response: "I'm good. You've already asked me",
        visit_count: Mutex::new(0),
        db: db_pool,
    });
    //Construct app and configure routes
    let app = move || {
        App::new()
            .app_data(shared_data.clone())
            .configure(general_routes)
            .configure(course_routes)
    };
    //Start HTTP server
    let host_port = env::var("HOST_PORT").expect("HOST:PORT addre
    HttpServer::new(app).bind(&host_port)? .run().await
}

```

Also ensure to add the environment variables for database access and server port numbers in .env file.

Do a sanity check by running the server with:

```
cargo run --bin iter4
```

This is the end state of Chapter 3, but recreated as the starting point for chapter 4.

Let's now take a quick tour of the basics of error handling in Rust, which we can then put to use for designing custom error handling for our web service.

5.2 Basics of error handling in Rust and Actix Web

Broadly, programming languages use one of two approaches for error handling - *exception handling* or *return value*. Rust uses the latter. This is different compared to languages like Java, Python or Javascript, where *exception handling* is used. In Rust, error handling is seen as an enabler of the reliability guarantees provided by the language, so Rust wants the programmer to handle errors explicitly rather than throw exceptions. Towards this goal, Rust functions that have a possibility of failures return a *Result enum* type whose definition is shown here:

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

A Rust function signature would contain a return value of type *Result<T,E>*, where T is the type of value that will be returned in a success case, and E is the type of Error value that will be returned in case of a failure. A *Result* type basically is a way of saying that a computation or function can return one of two possible outcomes, a value in case of a successful computation or an error in case of failures.

Let's see an example. Here is a simple function that parses a string into an integer , squares it and returns a value of type *i32*. If the parsing fails, it returns an error of type *ParseIntError*.

```
fn square(val: &str) -> Result<i32, ParseIntError> {
    match val.parse::<i32>() {
        Ok(num) => Ok(i32::pow(num, 2)),
        Err(e) => Err(e),
    }
}
```

Note that the parse function of the Rust standard library returns a *Result* type, which we are unwrapping (i.e. extracting value from) using a *match* statement. Note the return value from this function which is of the pattern *Result*<*T,E*> where, in this case, *T* is *i32* and *E* is *ParseIntError*.

Let's write a *main()* function that calls the *square()* function. Here is the complete code:

```
use std::num::ParseIntError;

fn main() {
    println!("{:?}", square("2"));
    println!("{:?}", square("INVALID"));
}

fn square(val: &str) -> Result<i32, ParseIntError> {
    match val.parse::<i32>() {
        Ok(num) => Ok(i32::pow(num, 2)),
        Err(e) => Err(e),
    }
}
```

Run this code and you will see the following output printed to the console.

```
Ok(4)
Err(ParseIntError { kind: InvalidDigit })
```

In the first case, the *square()* function is able to successfully parse the number 2 from the string, and returns the squared value enclosed in the *Ok()* enum type. In the second case, an error is returned of type *ParseIntError*, as the *parse()* function is unable to extract a number from the string.

Let's now look at a special operator that Rust provides to make error handling less verbose, the ? operator. Note that in the earlier code, we have used the *match* clause to unwrap the *Result* type returned from the *parse()* method. Let's next see usage of the ? operator to reduce boilerplate code:

```
use std::num::ParseIntError;

fn main() {
    println!("{:?}", square("2"));
    println!("{:?}", square("INVALID"));
}
```

```

fn square(val: &str) -> Result<i32, ParseIntError> {
    let num = val.parse::<i32>()?;
    Ok(i32::pow(num, 2))
}

```

You'll notice that the *match* statement with the associated clauses has been replaced by the `?` operator. This operator tries to unwrap the integer from the *Result* value and store it in the *num* variable. If unsuccessful, it receives the Error from the *parse()* method, aborts the *square* function and propagates the *ParseIntError* to the calling function (which in our case is the *main()* function).

We'll now take the next step to explore error handling in Rust, by adding additional functionality to the *square()* function. The code here shows additional lines of code to open a file and write the calculated square value to it.

```

use std::fs::File;
use std::io::Write;
use std::num::ParseIntError;

fn main() {
    println!("{}: {}", square("2"));
    println!("{}: {}", square("INVALID"));
}

fn square(val: &str) -> Result<i32, ParseIntError> {
    let num = val.parse::<i32>()?;
    let mut f = File::open("fictionalfile.txt")?;
    let string_to_write = format!("Square of {} is {}", num, i32::pow(num, 2));
    f.write_all(string_to_write.as_bytes())?;
    Ok(i32::pow(num, 2))
}

```

When you compile this code, you'll get an error message as follows:

```
the trait `std::convert::From<std::io::Error>` is not implemented
```

The error message may appear to be confusing, but what it's trying to say is that the *File::open* and *write_all* methods return a *Result* type containing an error of type *std::io::Error*, which should be propagated back to the *main()* function, as we have used the `?` operator. However, the function signature of

`square()` specifically states that it returns an error of type `ParseIntError`. We seem to have a problem now as there are two possible error types that can be returned from the function - `std::num::ParseIntError` and `std::io::Error`, but our function signature can only specify one error type.

This is where custom error types come in. Let's define a custom error type that can be an abstraction over the `ParseIntError` and `io::Error` types. Modify the code as shown:

```
use std::fmt;
use std::fs::File;
use std::io::Write;

#[derive(Debug)]
pub enum MyError {                                     #1
    ParseError,
    IOError,
}

impl std::error::Error for MyError {}                  #2

impl fmt::Display for MyError {                         #3
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match self {
            MyError::ParseError => write!(f, "Parse Error"),
            MyError::IOError => write!(f, "IO Error"),
        }
    }
}

fn main() {
    let result = square("INVALID");
    match result {                                     #4
        Ok(res) => println!("Result is {:?}", res),
        Err(e) => println!("Error in parsing: {:?}", e)
    };
}

fn square(val: &str) -> Result<i32, MyError> {
    let num = val.parse::<i32>().map_err(|_| MyError::ParseError)
    let mut f = File::open("fictionalfile.txt").map_err(|_| MyError::IOError)
    let string_to_write = format!("Square of {:?} is {:?}", num, num);
    f.write_all(string_to_write.as_bytes())
        .map_err(|_| MyError::IOError)?;
    Ok(i32::pow(num, 2))
}
```

```
}
```

We're making progress. We've so far seen how Rust uses *Result* type to return errors, how we can use the `? operator` to reduce boilerplate code to propagate errors, and how to define and implement custom error types to unify error handling at a function or application-level.

Rust's error handling makes code safe

A Rust function can either belong to the Rust standard library, an external crate or it can be a custom function written by the programmer. Whenever there is a possibility of error, Rust functions return a *Result* data type. The calling function must then handle the error either by a) *propagating the error* further to its caller using the `? operator`, b) *converting any errors received* into another type before bubbling it up, c) *handling the Result::Ok and Result::Error variants* using the `match` block, d) or *simply panic* on error with `.unwrap()` or `.expect()`. This makes programs safer because it is impossible to access invalid, null or uninitialized data that's returned from a Rust function.

Let's now take a look at how *Actix-web* builds on top of the Rust error-handling philosophy to return errors for web services and applications.

Figure 5.3. Converting errors to HTTP responses

How to convert Errors into HTTP responses?

ErrorResponse trait

Any error type that implements the *ErrorResponse* trait, can be converted into an HTTP Response message by Actix web

Built-in implementations

Actix-web contains default implementations of *ErrorResponse* trait for many common error types such as:

- Rust standard *I/O errors*,
- *Serde* errors,
- Actix web error types automatically implement this trait. Examples are *ProtocolError*, *Utf8Error*, *ParseError*, *ContentTypeError*, *PathError*, and *QueryPayloadError*.

Other error types

For any error types that need to be converted to HTTP responses, and for which default implementations (of *ErrorResponse* trait) are not available, custom implementations will have to be provided.

Actix-web has a general purpose error struct *actix_web::error::Error* which, like any other Rust error type, implements the Rust standard library's error trait *std::error::Error*. Any error type that implements the Rust standard library *Error* trait, can be converted into an Actix Error type with the `?` operator. The Actix Error type will then automatically be converted to an HTTP Response message that goes back to the HTTP client.

Here is an example of a basic *Actix handler* function that returns a *Result* type.

Create a new cargo project with *cargo new* and add the following to dependencies in *Cargo.toml*:

```
[dependencies]
actix-web = "3"
```

Add the following code to *src/main.rs*:

```
use actix_web::{error::Error, web, App, HttpResponse, HttpServer}

async fn hello() -> Result<HttpResponse, Error> {          #1
    Ok(HttpResponse::Ok().body("Hello there!"))           #2
}
```

```
#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().route("/hello", web::get().to(h
        .bind("127.0.0.1:3000")?
        .run()
        .await
    })
}
```

Even though the handler function signature specifies that it can return an Error type, the handler function is so simple that there is little possibility of anything going wrong here.

Run the program with:

```
cargo run
```

From a browser, connect to the *hello* route using:

```
http://localhost:3000/hello
```

You should see the following message displayed in your browser screen:

Hello there!

Now alter the handler function to include operations that can possibly fail.

```
use actix_web::{error::Error, web, App, HttpResponse, HttpServer}
use std::fs::File;
use std::io::Read;

async fn hello() -> Result<HttpResponse, Error> {
    let _ = File::open("fictionalfile.txt")?;
    Ok(HttpResponse::Ok().body("File read successfully"))
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    HttpServer::new(|| App::new().route("/hello", web::get().to(h
        .bind("127.0.0.1:3000")?
        .run()
        .await
    ))
}
```

Run the program again, and connect to the *hello* route from the browser. You should see the following message (or similar):

```
No such file or directory (os error 2)
```

To a discerning reader, two immediate questions may come to mind:

1. The file operation returns an error of type *std::io::Error*, as seen in the earlier example. How is it possible to send an error of type *std::io::Error* from the handler function, when the return type specified in function signature is *actix_web::error::Error*?
2. How did the browser display a text error message, when we returned an *Error* type from the handler function?

To answer the first question, anything that implements the *std::error::Error* trait (which the *std::io::Error* does), can be converted to *actix_web::error::error* type, as Actix framework implements the *std::error::Error* trait for its own type *actix_web::error::error*. This allows a question mark (?) to be used on the *std::io::Error* type to convert it into *actix_web::error::error* type. For reference , see this link (or a later version of this document available at the time when you are reading this):
https://docs.rs/actix-web/3.3.2/actix_web/error/struct.Error.html.

To answer the second question, anything that implements the Actix Web *ResponseError* trait can be converted to an HTTP response. Interestingly, the Actix-web framework contains built-in implementations of this trait for many common error types, and *std::io::Error* is one of them. For more details about available default implementations, refer to this link (or a later version of this document available at the time when you are reading this):
https://docs.rs/actix-web/3.3.2/actix_web/error/trait.ResponseError.html. The combination of *Actix Error* type and *ResponseError* trait provide a bulk of Actix's error handling support for web services and applications.

Getting back to our example, this is how, when an error of type *std::io::Error* is raised within the the *hello()* handler function, it gets eventually converted into an HTTP Response message.

We will utilize these feature of *Actix web* to convert a custom error type into

an HTTP Response message in this chapter.

With this background, you are now ready to start implementing Error handling in the tutor web service.

5.3 Defining a custom error handler

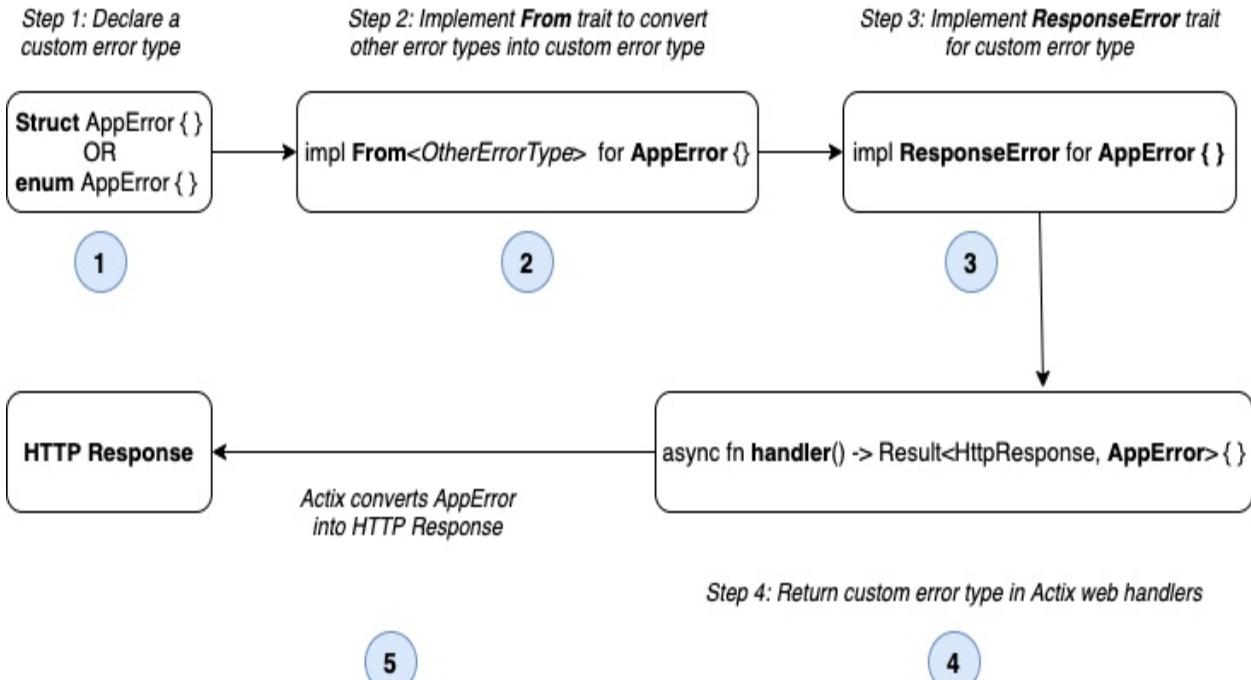
In this section, we'll define a custom error type for our web service. Before that, let's define the overall approach:

1. Define a custom error *enum* type that encapsulates the various types of errors that you expect to encounter within the web service.
2. Implement the *From* trait (from Rust standard library) to convert the other distinct error types into your custom error type.
3. Implement the Actix *ResponseError* trait for the custom error type. This enables Actix to convert the custom error into an HTTP response.
4. In the application code (e.g. handler functions), return the custom error type instead of standard Rust error type or Actix error type.
5. There is no step 5. Just sit back and watch Actix automatically converts any custom errors returned from the handler functions into valid HTTP Responses which are sent back to the client.

Figure 5.4 illustrates these steps.

Figure 5.4. Steps in writing a custom error type

Steps in writing a custom error type



That's it. Let's start by creating a new file `src/iter4/errors.rs`. We'll add the code for this file in three parts. Here is the code for part-1.

Listing 5.1. Error handling - part 1

```
use actix_web::{error, http::StatusCode, HttpResponse, Result};  
use serde::Serialize;  
use sqlx::error::Error as SQLxError;  
use std::fmt;  
  
#[derive(Debug, Serialize)]  
pub enum EzyTutorError {  
    DBError(String),  
    ActixError(String),  
    NotFound(String),  
}  
#[derive(Debug, Serialize)]  
pub struct MyErrorResponse {  
    error_message: String,  
}
```

We've defined two data structures for error handling - `EzyTutorError` which

is the primary error handling mechanism within the web service, and *MyErrorResponse* which is the user-facing message. To convert the former to the latter when an error occurs, let's write a method in the *impl* block of *EzyTutorError*.

Impl blocks

Just to recall, an *impl* block is Rust's way to allow developers to specify functions associated with a *data type*. This is the only way in Rust to define a function that can be invoked on an instance of the type in a *method-call* syntax. e.g. if *Foo* is the data type, *foo* is an instance of *Foo*, and *bar()* is the function defined within *impl* block of *Foo*, then the function *bar()* can be invoked on instance *foo* as follows: *foo.bar()*. *Impl* blocks also serve to group together functionality associated with a user-defined data type, that makes them easier to discover and in code maintenance. Further, *Impl* blocks allow the creation of *associated* functions which are basically functions associated with the *data type* rather than an *instance* of the *data type*. For example, to create a new instance of *Foo*, an associated function *new()* can be defined such that *Foo::new()* creates a new instance of *Foo*.

Listing 5.2. Error handling - part 2*

```
impl EzyTutorError {
    fn error_response(&self) -> String {
        match self {
            EzyTutorError::DBError(msg) => {
                println!("Database error occurred: {:?}", msg);
                "Database error".into()
            }
            EzyTutorError::ActixError(msg) => {
                println!("Server error occurred: {:?}", msg);
                "Internal server error".into()
            }
            EzyTutorError::NotFound(msg) => {
                println!("Not found error occurred: {:?}", msg);
                msg.into()
            }
        }
    }
}
```

We have defined a method called `error_response()` on our custom error struct `EzyTutorError`. This method will be called when we want to send a user-friendly message to inform the user that an error has occurred. Here we are handling all three types of errors, with the goal of sending back a simpler, friendly error message to the user.

We have so far defined error data structures and even written a method to convert custom error struct to a user friendly text message. The question that arises is how can we propagate an error to an HTTP client from the web service? The only way an HTTP web service can communicate with a client is through an HTTP response message, right?

So, what's missing is a way to convert the custom error that is generated in the server into a corresponding HTTP response message. We've seen in the earlier example how to achieve this using the `actix_web::error::ResponseError` trait. If a handler returns an error that also implements `ResponseError` trait, `Actix web` will convert that error into an HTTP response, with the corresponding status code.

In our case, this boils down to implementing the `ResponseError` trait on the `EzyTutorError` struct. To implement this trait means to implement two methods defined on the trait- `error_response()` and `status_code`. Let's look at the code:

Listing 5.3. Error handling - part 3

```
impl error::ResponseError for EzyTutorError {
    fn status_code(&self) -> StatusCode { #1
        match self {
            EzyTutorError::DBError(msg) | EzyTutorError::ActixError
                StatusCode::INTERNAL_SERVER_ERROR
            }
            EzyTutorError::NotFound(msg) => StatusCode::NOT_FOUND,
        }
    }
    fn error_response(&self) -> HttpResponse { #2
        HttpResponse::build(self.status_code()).json(MyErrorResponse {
            error_message: self.error_response(),
        })
    }
}
```

Now that we've defined the custom error type , let's next incorporate this into the handler and database access code for the three APIs of the web service.

5.4 Error handling for retrieving all courses

In this section, we'll incorporate error handling for the API to retrieve the course list for a tutor. Let's focus on the file *db_access.rs* which contains functions for database access.

Add the following import to this file (*db_access.rs*):

```
use super::errors::EzyTutorError;
```

The *super* keyword refers to the parent scope (for *db_access* module), which is where the *errors* module is located.

Let's look at a chunk of the existing code in the function *get_courses_for_tutor_db*.

```
let course_rows = sqlx::query!(
    "SELECT tutor_id, course_id, course_name, posted_time FROM
     tutor_id
)
.fetch_all(pool)
.await.?;
.unwrap();
```

Note in particular the *unwrap()* method . This is a short-cut to handle errors in Rust. Whenever an error occurs in the database operation, the program thread would panic and exit. The *unwrap()* keyword in Rust means "if the operation is successful, return the result, which in this case is the list of courses. In case of error just panic and abort the program".

This was alright so far, as we were just learning how to build the web service. But this is not the expected behaviour for a production service. We cannot allow the program execution to panic and exit, for every error in database access. What we want to do instead is to handle the error in some way. If we know what to do with the error itself, we can do it there. Otherwise, we can propagate the error from the *database access* code to the calling *handler*

function, which can then figure out what to do with the error. To achieve this propagation, we can use the question mark operator (?) instead of the *unwrap()* keyword, as shown.

```
let course_rows = sqlx::query!(
    "SELECT tutor_id, course_id, course_name, posted_time FROM
    tutor_id
)
.fetch_all(pool)
.await?;
```

Note that the *.unwrap()* method, which operates on the result of the database fetch operation, has now been replaced with a question mark (?). While the earlier *unwrap()* operation told the Rust compiler to panic in case of errors, the ? tells the Rust compiler that "in case of errors, convert the sqlx database error into another error type and return from the function, propagating the error to the calling handler function". The question now is, to what type would the question mark operator convert the database error?

We'd have to specify that.

In order to propagate the error in this manner (using ?), we need to alter the database method signature to return a *Result* type. As we've seen earlier, a *Result* type expresses the possibility of an error. It provides a way to represent one out of two possible outcomes in any computation or function call - *Ok(val)* in case of success where *val* is the result of the successful computation, or *Err(err)* in case of errors where *err* is the error returned from the computation.

In our database fetch function, let's define these two possible outcomes as the following:

- return a Vector of courses - *Vec<Course>*, in case database access is successful, or
- return an error of type *EzyTutorError* in case the database fetch fails.

If we revisit the *await?* expression at the end of the database fetch operation, we can interpret it to mean that if the database access fails, convert the sqlx database error into an error of type *EzyTutorError*, and return from the

function. In such a case of failure, the calling handler function would receive back an error of type *EzyTutorError* from the database access function.

Here is the modified code in *db_access.rs*. The changes are highlighted in the numbered annotations.

Listing 5.4. Error handling in database access method to retrieve courses for tutor

```
pub async fn get_courses_for_tutor_db(
    pool: &PgPool,
    tutor_id: i32,
) -> Result<Vec<Course>, EzyTutorError> {                      #1
    // Prepare SQL statement
    let course_rows = sqlx::query!(
        "SELECT tutor_id, course_id, course_name, posted_time FROM
        tutor_id
    )
    .fetch_all(pool)
    .await?;                                #2
    // Extract result

    let courses: Vec<Course> = course_rows
        .iter()
        .map(|course_row| Course {
            course_id: course_row.course_id,
            tutor_id: course_row.tutor_id,
            course_name: course_row.course_name.clone(),
            posted_time: Some(chrono::NaiveDateTime::from(course_
        })
        .collect();
    match courses.len() {                      #3
        0 => Err(EzyTutorError::NotFound(
            "Courses not found for tutor".into(),
        )), 
        _ => Ok(courses),
    }
}
```

The last point about returning an error in case no courses are found for a valid *tutor id* can be debated as to whether it really is an error. Let's however just set this argument aside for now, and use this as another opportunity to practice error handling in Rust.

Let's also alter the calling handler function (in *iter4/handler.rs*) to

incorporate error handling. First add the following import:

```
use super::errors::EzyTutorError;
```

Modify the `get_courses_for_tutor()` function to return a `Result` type:

```
pub async fn get_courses_for_tutor(
    app_state: web::Data<AppState>,
    path: web::Path<i32>,
) -> Result<HttpResponse, EzyTutorError> {
    let tutor_id = path.into_inner();
    get_courses_for_tutor_db(&app_state.db, tutor_id)
        .await
        .map(|courses| HttpResponse::Ok().json(courses))
}
```

It appears that we've completed the error handling implementation for retrieving course lists. Compile and run the code with:

```
cargo run --bin iter4
```

You will notice there are compiler errors.

This is because, for the `? operator` to work , each error raised in the program should be converted first to type `EzyTutorError`. For example, if there is an error in database access using `sqlx`, `sqlx` returns an error of type `sqlx::error::DatabaseError` and Actix does not know how to deal with it. So, we must tell Actix how to convert the `sqlx` error to our custom error type `EzyTutorError`. Did you actually think Actix will do it for you? Sorry, you have to write the code!

The code shown here is to be added to `iter4/errors.rs`.

Listing 5.5. Implementing From and Display traits for EzyTutorError

```
impl fmt::Display for EzyTutorError {           #1
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{}", self)
    }
}

impl From<actix_web::error::Error> for EzyTutorError { #2
```

```

    fn from(err: actix_web::error::Error) -> Self {
        EzyTutorError::ActixError(err.to_string())
    }
}

impl From<SQLxError> for EzyTutorError { #3
    fn from(err: SQLxError) -> Self {
        EzyTutorError::DBError(err.to_string())
    }
}

```

We have now made the necessary changes to both the *database access* code and the *handler* code, to incorporate error handling for retrieving course lists. Build and run the code with:

```
cargo run --bin iter4
```

From a browser access the following URL:

```
http://localhost:3000/courses/1
```

You should be able to see the list of courses.

Let's test the error conditions now.

Access the API with an invalid *tutor id* as shown:

```
http://localhost:3000/courses/10
```

You should see the following displayed in the browser:

```
{"error_message": "Courses not found for tutor"}
```

This is as intended.

Let's now try simulating another type of error . This time we will simulate an error in sqlx database access.

In the *.env* file change the database URL to an invalid user id. An example is shown below:

```
DATABASE_URL=postgres://invaliduser:trupwd@127.0.0.1:5432/truwitt
```

Restart the web service with

```
cargo run --bin iter4
```

Access the valid URL as shown:

```
http://localhost:3000/courses/1
```

You should see the following error message in the browser:

```
{"error_message": "Database error"}
```

Let's spend a few minutes understanding what happened here.

When we provided an invalid database url, on receipt of the API request, the web service database access function tried to create a connection from the connection pool and run the query. This operation failed and an error of type `sqlx::error::DatabaseError` was raised by the `sqlx` client. This error was converted to our custom error type `EzyTutorError` due to the following `From` trait implementation in `errors.rs`

```
impl From<SQLxError> for EzyTutorError { }
```

The error of type `EzyTutorError` was then propagated from the database access function in `db_access.rs` to the handler function in `handlers.rs`. On receipt of this error, the handler function propagates it further to the Actix web framework, which then converts this error into an HTML response message with an appropriate error message.

Now, how do we check this error status code? This can be verified by accessing the URL using a command-line HTTP client. We'll use `curl` with the verbose option as follows:

```
curl -v http://localhost:3000/courses/1
```

You should see a message in your terminal, similar to that shown here:

```
GET /courses/1 HTTP/1.1
```

```
> Host
```

```
: localhost:3000
```

```
> User-Agent: curl/7.64.1
> Accept: /*
>
< HTTP/1.1 500 Internal Server Error
```

Go back to the function `status_code()` in `iter4/errors.rs`. You'll notice that for database and actix errors, we are returning a status code of `StatusCode::INTERNAL_SERVER_ERROR`, which translates to an HTML response status code of `500`. This matches the output generated by the `curl` tool.

Before we move on, make sure you correct the database URL username to the right value in the `.env` file, otherwise future tests will fail.

We have thus implemented custom error handling for the first API. Let's also ensure that the test scripts are not broken. Run the tests as follows:

```
cargo test --bin iter4
```

You will find that the compiler throws errors. This is because our test script also must be modified to receive an error response from the handler. Make changes to the test script in `handlers.rs` as shown:

Listing 5.6. Test script for getting all courses for tutor

```
#[actix_rt::test]
async fn get_all_courses_success() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    });
    let tutor_id: web::Path<i32> = web::Path::from(1);
    let resp = get_courses_for_tutor(app_state, tutor_id).awa
    assert_eq!(resp.status(), StatusCode::OK);
}
```

Note: Actix web does not support propagating errors using the question mark (?) operator, so we have to use `unwrap()` or `expect()` to extract the HTTP

response from the *Result* type.

Rerun the following command from the command-line:

```
cargo test get_all_courses_success --bin iter4
```

You should see the tests successfully run.

You'll notice that in the previous command, we ran only the specific test case *get_all_courses_success*. In case you run the entire test suite with *cargo test --bin iter4*, you may get an error similar to this:

```
DBError("duplicate key value violates unique constraint")
```

This is because everytime the test suite is run, a new record with *course_id* = 3 is inserted into the table. If the tests are run the second time, this insertion of record fails as *course_id* is the primary key in table, and there cannot be two records with the same *course_id*. In such a case, simply login to the psql shell and delete the entry with *course_id* = 3 from the table *ezy_course_c5*.

There is a simpler option though. You can tell the cargo test executor to ignore any specific test case in the test suite with the *#[ignore]* annotation. You can specify this annotation as shown:

```
#[ignore]
#[actix_rt::test]
async fn post_course_success() {
```

Now, you can run the entire test suite with *cargo test --bin iter4*, and you will see something similar to this printed on your console:

```
running 3 tests
test handlers::tests::post_course_success ... ignored
test handlers::tests::get_all_courses_success ... ok
test handlers::tests::get_course_detail_test ... ok

test result: ok. 2 passed; 0 failed; 1 ignored; 0 measured; 0 fil
```

You'll notice that the *post_course_success* test case has been ignored and the other two tests have been run.

We now have to perform the same steps for the other two APIs also, i.e., change database access functions, handler methods and test scripts.

5.5 Error handling for retrieving course details

Let's look at the changes needed to incorporate error handling for the second API, i.e., getting course details.

Here is the updated database access code in `db_access.rs`:

Listing 5.7. Error handling in function to get course details

```
pub async fn get_course_details_db(pool: &PgPool, tutor_id: i32,
    // Prepare SQL statement
    let course_row = sqlx::query!(
        "SELECT tutor_id, course_id, course_name, posted_time FROM
        tutor_id, course_id
    )
    .fetch_one(pool)
    .await;
    if let Ok(course_row) = course_row {           #2
        // Execute query
        Ok(Course {
            course_id: course_row.course_id,
            tutor_id: course_row.tutor_id,
            course_name: course_row.course_name.clone(),
            posted_time: Some(chrono::NaiveDateTime::from(course_row.
        })
    } else {
        Err(EzyTutorError::NotFound("Course id not found".into()))
    }
}
```

Let's update the handler function:

```
pub async fn get_course_details(
    app_state: web::Data<AppState>,
    path: web::Path<(i32, i32)>,
) -> Result<HttpResponse, EzyTutorError> {           #1
    let (tutor_id, course_id) = path.into_inner();
    get_course_details_db(&app_state.db, tutor_id, course_id)
        .await
        .map(|course| HttpResponse::Ok().json(course))      #2
}
```

```
}
```

Restart the web service with

```
cargo run --bin iter4
```

Access the valid URL as shown:

```
http://localhost:3000/courses/1/2
```

You will see the course details displayed as before. Now try accessing details for an invalid course id:

```
http://localhost:3000/courses/1/10
```

You should see the following error message in the browser:

```
{"error_message": "Course id not found"}
```

Let's also alter the test script *async fn get_course_detail_test()* in *handlers.rs* to accommodate errors returned from the handler function.

```
let resp = get_course_details(app_state, parameters).await.unwrap
```

Run the test with:

```
cargo test get_course_detail_test --bin iter4
```

The test should pass.

Next, we'll incorporate error handling for posting a new course.

5.6 Error handling for posting a new course

We'll basically follow the same set of steps like for the other two APIs, i.e. modify the database access function, the handler function and the test script.

Let's start with the database access function in *db_access.rs*.

Listing 5.8. Error handling in database access function to post new course

```

pub async fn post_new_course_db(
    pool: &PgPool,
    new_course: Course,
) -> Result<Course, EzyTutorError> { #1
    let course_row = sqlx::query!("insert into ezy_course_c5 (cou
        .fetch_one(pool)
        .await?; #2
    //Retrieve result
    Ok(Course { #3
        course_id: course_row.course_id,
        tutor_id: course_row.tutor_id,
        course_name: course_row.course_name.clone(),
        posted_time: Some(chrono::NaiveDateTime::from(course_row.
    })
}

```

Update the handler function:

```

pub async fn post_new_course(
    new_course: web::Json<Course>,
    app_state: web::Data<AppState>,
) -> Result<HttpResponse, EzyTutorError> { #1
    post_new_course_db(&app_state.db, new_course.into())
        .await
        .map(|course| HttpResponse::Ok().json(course)) #2
}

```

Finally, update the test script `async fn post_course_success()` in `handlers.rs` to add `unwrap()` on the return value from database access function, as shown:

```

#[actix_rt::test]
async fn post_course_success() {
    /// all code not shown here
    let resp = post_new_course(course_param, app_state).await
#1
    assert_eq!(resp.status(), StatusCode::OK);
}

```

Rebuild and restart the web service with:

```
cargo run --bin iter4
```

Post a new course from command line with:

```
curl -X POST localhost:3000/courses/ -H "Content-Type: applicatio
```

Verify that the new course has been added with the following URL on the browser:

```
http://localhost:3000/courses/1/4
```

Run the tests with:

```
cargo test --bin iter4
```

All three tests should successfully pass.

Let's do a quick recap. In this chapter, you have learnt how to transform different types of errors encountered in the web service into a custom error type, and how to transform that into an HTTP Response message, thus providing a meaningful message to the client in case of server errors. Along the way, you have also picked up finer concepts of error handling in Rust that can be applied to any Rust application. More importantly, you now know how to handle failures gracefully, provide meaningful feedback to users, and build a solid and stable web service.

With this, you have also completed the implementation of error handling for the three APIs for the tutor web service. The web service is backed by a database and can handle database and actix errors, and also invalid inputs from users. Congratulations!

5.7 Summary

1. Rust provides a robust and ergonomic error handling approach with features such as *Result* type, combinator functions such as *map* and *map_err* that operate on the *Result* type, quick code prototyping options *withunwrap()* and *expect()*, the *?* operator to reduce code boilerplate, and the ability to convert errors from one error type to another using the *From* trait.
2. *Actix web* builds on top of Rust's error handling features to include its own *Error* type and the *ResponseError* trait. These enable Rust programmers to define custom error types and have the *Actix web* framework automatically convert them into meaningful HTTP response messages at runtime, for sending back to the web client or user. Further,

Actix web provides built-in *From* implementations to convert Rust standard library error types into *Actix Error* type, and also provides default *ResponseError* trait implementations to convert Rust standard library error types into HTTP Response messages.

3. Implementing custom error handling in *Actix* involves the following steps:
 - Define a data structure to represent a custom error type,
 - Define possible values the custom error type can take (for example, database errors, not found errors etc)
 - Implement *ResponseError* trait on the custom error type
 - Implement *From* traits to convert various types of errors (such as *sqlx* errors or *Actix web* errors) into custom error type
 - Change the return values of *database access* functions and *route handler* functions, to return the custom error type in case of errors. The *Actix web* framework then converts the custom error type into an appropriate HTTP response, and embeds the error message within the body of the HTTP response.
4. In this chapter, custom error handling was incorporated for each of the three APIs in the *tutor* web service.

Our *tutor* web service is now functional with a full-fledged database to persist data, and a robust error handling framework that can be customized further as the features evolve. In the next chapter, we will deal with another typical real-world situation, i.e. changes in product requirements from management team, and additional feature requests from users. Will Rust stand up to the test of large-scale refactoring of code?

Switch to the next chapter to find out.

6 Evolving the APIs and fearless refactoring

This chapter covers

- Revamping the project structure
- Enhancing the data model for course creation & management
- Enabling tutor registration and management

In the previous chapter, we covered the basics of error handling in Rust and how we can design custom error handling for our web service. After working through the last few chapters, you should by now have the foundational understanding of how a web service is structured using the Actix web framework, how you can talk to a relational database for CRUD activities, and how to handle any errors that occur while processing incoming data and requests.

In this chapter, we will step up the pace and deal with something that we cannot avoid in the real-world: *changes*.

Every actively-used web service or application evolves significantly over its lifecycle, based on user feedback or business requirements. And many of these newer requirements could mean breaking changes to the web service/application. In this chapter, you'll learn how Rust helps you cope with situations involving drastic changes in the design and rewriting significant parts of your existing code. You'll use the power of the Rust compiler and the features of the language, to come out of this challenge with a smile on your face.

In this chapter you will fearlessly take on several changes to the web service. You'll redesign the data model for courses, add course routes, modify handler and database access functions and update the test cases. You'll also design and build a new module in the application to manage tutor information, and to define the relationship between tutors and courses. You'll enhance the

error handling features of the web service to cover edge cases. If this isn't enough, you'll also fully revamp the project code and directory structure to neatly segregate code across Rust modules.

There's no time to waste, let's get going.

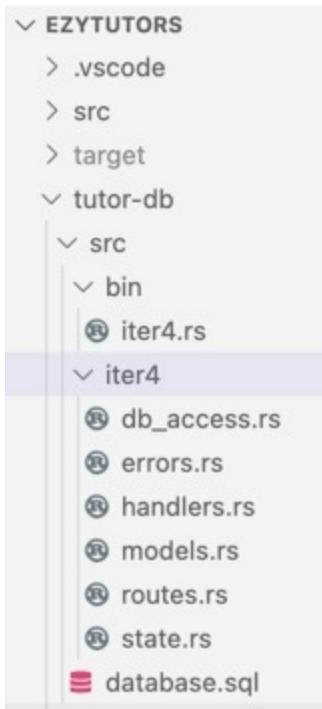
6.1 Revamping the project structure

In the previous chapter, we focused on creating and maintaining basic course data. In this chapter we'll enhance the course module, and also add functionality to create and maintain tutor information. As the size of the codebase will grow, this is a good time to rethink the project structure. So in this section, we'll start by reorganizing the project into a structure that aids in code development and maintenance as the application becomes larger and more complex.

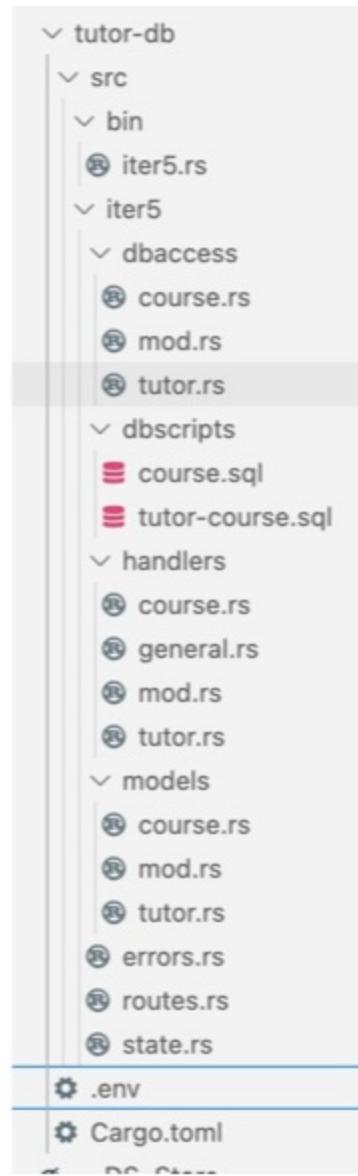
Figure 6.1 shows two views. On the left is the project structure that we'll start with. On the right is the structure that we'll end up with.

Figure 6.1. Project structure

Project structure in chapter 5



Project structure in chapter 6



The main change you will notice is that in the proposed project structure, *dbaccess*, *handlers* and *models* are not single files, but represent folders. The database access code for *Course* and *Tutor* will be organized under *dbaccess* folder. Likewise, for *models* and *handlers*. This approach reduces the length of individual files, while making it quicker to navigate to what you are looking for, though it adds some complexity to the project structure.

Before we begin, let's set up the PROJECT_ROOT environment variable to point to the full path of the project root(*ezytutors/tutor_db*).

```
export PROJECT_ROOT=<full-path-to ezytutors/tutor-db folder>
```

Verify that it is set correctly using

```
echo $PROJECT_ROOT
```

Henceforth, the term *project root* would refer to the folder path stored in \$PROJECT_ROOT environment variable. References to other files in this chapter will be made with respect to the *project root*.

The code structure is described here:

1. **\$PROJECT_ROOT/src/bin/iter5.rs** : *main()* function.
2. **\$PROJECT_ROOT/src/iter5/routes.rs**: Contains routes. This will continue to be a single file containing all routes
3. **\$PROJECT_ROOT/src/iter5/state.rs**: Application state containing the dependencies that are injected into each thread of application execution.
4. **\$PROJECT_ROOT/src/iter5/errors.rs**: Custom error data structure and associated error handling functions
5. **\$PROJECT_ROOT/.env**: Environment variables containing database access credentials. This file should not be checked into the code repository.
6. **\$PROJECT_ROOT/src/iter5/dbscripts**: Database tables creation scripts for postgres.
7. **\$PROJECT_ROOT/src/iter5/handlers**:
 - A. **\$PROJECT_ROOT/src/iter5/handlers/course.rs**: Course-related handler functions
 - B. **\$PROJECT_ROOT/src/iter5/handlers/tutor.rs**: Tutor-related handler functions
 - C. **\$PROJECT_ROOT/src/iter5/handlers/general.rs**: Health check handler function
 - D. **\$PROJECT_ROOT/src/iter5/handlers/mod.rs**: Converting the directory *handlers* into a Rust module, so the Rust compiler knows how to find the dependent files.
8. **\$PROJECT_ROOT/src/iter5/models**:
 - A. **\$PROJECT_ROOT/src/iter5/models/course.rs**: Course-related data structures and utility methods
 - B. **\$PROJECT_ROOT/src/iter5/models/tutor.rs**: Tutor-related data

structures and utility methods

C. `$PROJECT_ROOT/src/iter5/models/mod.rs`: Converting the directory *models* into a Rust module, so the Rust compiler knows how to find the dependent files.

9. **`$PROJECT_ROOT/src/iter5/dbaccess`:**

A. `$PROJECT_ROOT/src/iter5/dbaccess/course.rs`: Course-related database-access methods

B. `$PROJECT_ROOT/src/iter5/dbaccess/tutor.rs`: Tutor-related database-access methods

C. `$PROJECT_ROOT/src/iter5/dbaccess/mod.rs`: Converting the directory *dbaccess* into a Rust module, so the Rust compiler knows how to find the dependent files.

Copy the code from chapter5's *iter4* folder as the starting point for this chapter.

Without adding any new functionality, let's just reorganize the existing code of chapter 5, into this new project structure.

Start with the following steps:

1. Rename `$PROJECT_ROOT/src/bin/iter4.rs` to `$PROJECT_ROOT/src/bin/iter5.rs`
2. Rename `$PROJECT_ROOT/src/iter4` folder to `$PROJECT_ROOT/src/iter5`
3. Under `$PROJECT_ROOT/src/iter5` create three subfolders *dbaccess*, *models* and *handlers_*.
4. Move and rename `$PROJECT_ROOT/src/iter5/models.rs` to `$PROJECT_ROOT/src/iter5/models/course.rs`
5. Create two more files under `$PROJECT_ROOT/src/iter5/models` folder - *tutor.rs* and *mod.rs*. Leave both files blank for now.
6. Move and rename `$PROJECT_ROOT/src/iter5/dbaccess.rs` to `$PROJECT_ROOT/src/iter5/dbaccess/course.rs`.
7. Create two more files under `$PROJECT_ROOT/src/iter5/dbaccess` folder - *tutor.rs* and *mod.rs*. Leave both files blank for now.
8. Move and rename `$PROJECT_ROOT/src/iter5/handlers.rs` to `$PROJECT_ROOT/src/iter5/handlers/course.rs`.
9. Create three more files under `$PROJECT_ROOT/src/iter5/handlers`

folder - *tutor.rs*, *general.rs* and *mod.rs*. Leave all three files blank for now.

10. Create a folder *\$PROJECT_ROOT/src/iter5/dbscripts*. Move and rename the existing *database.sql* file in the project folder to this directory, and rename it as *course.sql*. We'll modify this file later.

At this stage, ensure that your project structure looks similar to that shown in figure 6.1. Now that we have the project folder structure in place, let's modify the existing code to align to this new structure.

11. In the *mod.rs* file under *\$PROJECT_ROOT/src/iter5/dbaccess* and *\$PROJECT_ROOT/src/iter5/models* folders, add the following code.

```
pub mod course;  
pub mod tutor;
```

This tells the Rust compiler to consider the contents of the folders *\$PROJECT_ROOT/src/iter5/models* and *\$PROJECT_ROOT/src/iter5/dbaccess* as Rust modules. This allows us to, for example, refer and use the *Course* data structure in another source file like this. Note the similarity between the folder structure and module organisation.

```
use crate::models::course::Course;
```

12. Similarly, in the *mod.rs* file under *\$PROJECT_ROOT/src/iter5/handlers*, add the following code.

```
pub mod course;  
pub mod tutor;  
pub mod general;
```

13. Add the following imports to *\$PROJECT_ROOT/src/iter5/handlers/general.rs*:

```
use super::errors::EzyTutorError;  
use super::state::AppState;  
use actix_web::{web, HttpResponse};
```

Further move the function *pub async fn health_check_handler() {}* from

`$PROJECT_ROOT/src/iter5/handlers/course.rs` to
`$PROJECT_ROOT/src/iter5/handlers/general.rs`.

14. Let's now move to the `main()` function. In `$PROJECT_ROOT/src/bin/iter5`, adjust the module declaration paths to look like this:

```
#[path = "../iter5/dbaccess/mod.rs"]
mod dbaccess;
#[path = "../iter5/errors.rs"]
mod errors;
#[path = "../iter5/handlers/mod.rs"]
mod handlers;
#[path = "../iter5/models/mod.rs"]
mod models;
#[path = "../iter5/routes.rs"]
mod routes;
#[path = "../iter5/state.rs"]
mod state;
```

15. Adjust the module import paths in `$PROJECT_ROOT/src/iter5/dbaccess/course.rs` as shown:

```
use crate::errors::EzyTutorError;
use crate::models::course;
```

16. Adjust the module import paths in `$PROJECT_ROOT/src/iter5/handlers/course.rs` as shown:

```
use crate::dbaccess::course::*;
use crate::errors::EzyTutorError;
use crate::models::course;
```

17. Lastly, adjust the module paths in `$PROJECT_ROOT/src/iter5/routes.rs` as shown:

```
use crate::handlers::{course::*, general::*};
```

In this code refactoring exercise, ensure you do not delete any of the other import statements already existing, such as those related to Actix web. These are not mentioned because there is no change to their module paths.

Now from the *project root* check for compilation errors with the following command:

```
cargo check
```

You can also run the test script, which should execute successfully:

```
cargo test
```

If there are still any errors, revisit the steps. Otherwise, you should see the compilation going through successfully. Congratulations, you've successfully completed the refactoring of the project code into the new structure.

By way of recap, what we have done is that the code has been split into multiple smaller files, each performing a specific function (in line with *single responsibility principle* in software engineering). Secondly, we have grouped related files under common folders. For example, the database access code for *tutors* and *courses* is in separate source files, while both the source files are together placed under a *dbaccess* folder. We have clearly separated the namespaces (through use of Rust modules) for handler functions, database access, data model, routes, errors, database scripts, application state and error handling. This kind of intuitive project structure and file-naming enables collaboration among multiple developers that are involved in reviewing and modifying a code repository, speeds up ramp-up time for new team member onboarding, and reduces time-to-release for defect fixes and code enhancements. But this type of structure could be an overkill for small projects, so decisions on refactoring code should be made based on how code and functional complexity evolves over time.

We can now focus on functionality enhancements starting from the next section.

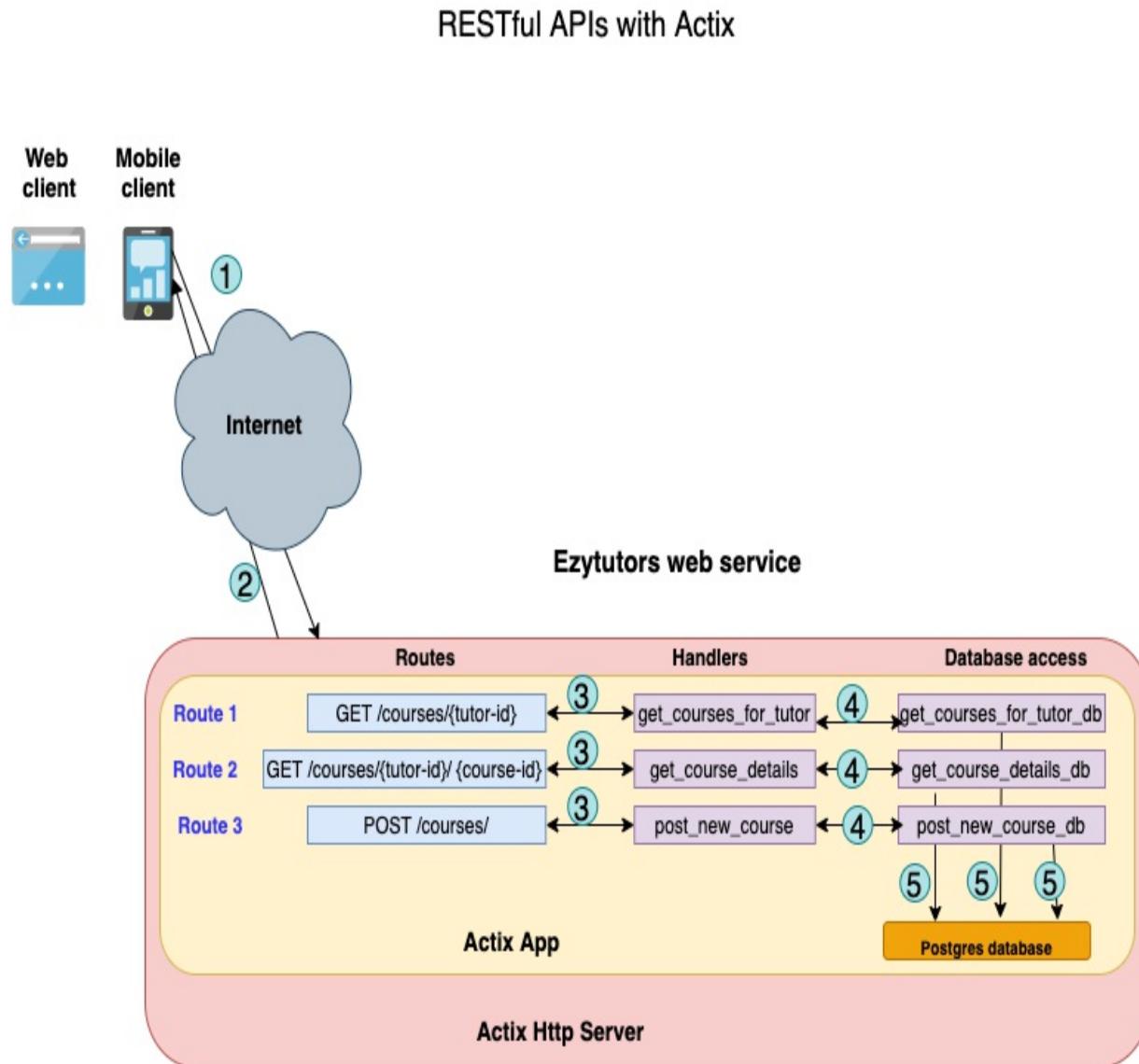
6.2 Enhancing the data model for course creation & management

In this section, we'll enhance the course-related APIs which will involve changes to the Rust data model, database table structure, routes, handlers and

database access functions.

Figure 6.2 shows the final code structure for course-related APIs. In the figure, the course-related API routes are listed, along with the names of the respective handler functions and database access functions.

Figure 6.2. Code structure for course-related APIs



Note the general naming convention followed for the database access functions, which are named by using the corresponding handler function name, and suffixing *db* to it.

Let's start by looking at the current Course data model in `$PROJECT_ROOT/src/iter5/models/course.rs`.

```
pub struct Course {  
    pub course_id: i32,  
    pub tutor_id: i32,  
    pub course_name: String,  
    pub posted_time: Option<NaiveDateTime>,  
}
```

This data structure has served its purpose until now, but it is elementary. It's time to add more real-world attributes to describe a course. Let's enhance the `Course` struct to add the following details:

- **Description:** Textual information describing the course so prospective students can decide if the course is for them
- **Format:** The course can be delivered in multiple-formats such as self-paced video course, e-book format or as instructor-led in-person training.
- **Structure of course:** We'll right now allow the tutor to upload a document that describes the course (such as a brochure in pdf format).
- **Duration of course:** Length of the course. This is typically described in terms of duration of video recording for video-based courses, duration of in-person training hours, or recommended study hours in case of e-books.
- **Price:** Specify the course price in US dollars
- **Language:** Since we expect to have an international audience for the web app, let's also allow courses in multiple languages.
- **Level:** This denotes the level of the student the course is targeted at. Possible values include *Beginner*, *Intermediate* and *Expert*.

In the next subsection, let's make the actual changes to the Rust data model.

6.2.1 Making changes to the data model

Let's begin the changes, starting with file imports.

Here is the original set of imports:

```
use actix_web::web;
use chrono::NaiveDateTime;
use serde::{Deserialize, Serialize};
```

Let's now alter the *Course* data structure to incorporate the additional data elements that we wish to capture.

Here is the updated *Course* data structure in
\$PROJECT_ROOT/src/iter5/models/course.rs:

```
#[derive(Serialize, Debug, Clone, sqlx::FromRow)]
pub struct Course {
    pub course_id: i32,
    pub tutor_id: i32,
    pub course_name: String,
    pub course_description: Option<String>,
    pub course_format: Option<String>,
    pub course_structure: Option<String>,
    pub course_duration: Option<String>,
    pub course_price: Option<i32>,
    pub course_language: Option<String>,
    pub course_level: Option<String>,
    pub posted_time: Option<NaiveDateTime>,
}
```

Note that we've declared a struct that has three mandatory fields - `course_id`, `tutor_id` and `course_name`, and the rest are optional (denoted by `Option<T>` type). This is to reflect the possibility that a course record in the database may not have values for these optional fields.

We've also auto-derived a few traits. `Serialize` is to be able to send the fields of `course` struct back to the api client. `Debug` is to enable printing of struct values during the development cycle. `Clone` is to help us duplicate string values, while complying with the Rust ownership model. `sqlx::FromRow` is to enable automatic conversion of a database record into the `Course` struct, while reading values from the database. We'll see how to implement this feature when we write the database access functions.

If we look at the *Course* data structure, there are a couple of fields- *posted time*, and *course id* which we plan to auto-generate at the database-level. So, while we need these fields to fully represent a *Course* record, we don't need these to be sent by an api client, in order to create a new course. So, how do

we handle these different representations of a *Course*?

Let's create a separate data structure that would only contain the fields relevant for the front-end for creation of a new course.

Here is the new struct *CreateCourse*:

```
#[derive(Deserialize, Debug, Clone)]
pub struct CreateCourse {
    pub tutor_id: i32,
    pub course_name: String,
    pub course_description: Option<String>,
    pub course_format: Option<String>,
    pub course_structure: Option<String>,
    pub course_duration: Option<String>,
    pub course_price: Option<i32>,
    pub course_language: Option<String>,
    pub course_level: Option<String>,
}
```

From this struct, we are conveying the intent that for creating a new course, `tutor_id` and `course_id` are mandatory fields, and the rest are optional, as far as the api client is concerned. But note that for the tutor web service, both `course_id` and `posted_time` are also mandatory fields for creating a new course, but these will be auto-generated internally.

You'll also notice that we've auto-derived `Deserialize` trait for *CreateCourse*, whereas we had auto-derived `Serialize` trait for *Course* struct. Why do you think we've done this?

This is because the *CreateCourse* struct will be used as the data structure to carry inputs from the user to the web service as part of the HTTP request body. Hence, the Actix-web framework needs to have a way to deserialize the data coming in over the wire into the *CreateCourse* Rust struct. (Note that for HTTP requests, the API client *serializes* the data payload for transmission, while the Actix-framework at the receiving end will *deserialize* the data back into a suitable form for processing by application).

To be precise, the Actix web framework serializes the incoming data payload into an Actix data type `web::Json<CreateCourse>`, but our application does not understand this type. So, we'll have to convert this Actix type into a

regular Rust struct. We'll implement the Rust *From* trait to write the conversion function, which we can then invoke at run-time, whenever a new HTTP request is received to create a new course.

```
impl From<web::Json<CreateCourse>> for CreateCourse {  
    fn from(new_course: web::Json<CreateCourse>) -> Self {  
        CreateCourse {  
            tutor_id: new_course.tutor_id,  
            course_name: new_course.course_name.clone(),  
            course_description: new_course.course_description.clo  
            course_format: new_course.course_format.clone(),  
            course_structure: new_course.course_structure.clone()  
            course_level: new_course.course_level.clone(),  
            course_duration: new_course.course_duration.clone(),  
            course_language: new_course.course_language.clone(),  
            course_price: new_course.course_price,  
        }  
    }  
}
```

Note that this conversion is relatively straight-forward , however if there is a possibility of errors during conversion, we would use the *TryFrom* trait, instead of the *From* trait. Errors can occur, for example, if we call any Rust standard lib function that returns a *Result* type, such as converting a string value to an integer.

The same conversion function implementing the *TryFrom* trait is shown here just as a reference:

First you'll need to import the *TryFrom* trait from the Rust standard library.

```
use std::convert::TryFrom;
```

Then you'll need to implement the *try_from* function and declare the type for Error that will be returned in case of problems in processing.

```
impl TryFrom<web::Json<CreateCourse>> for CreateCourse {  
    type Error = EzyTutorError;  
  
    fn try_from(new_course: web::Json<CreateCourse>) -> Result<Se  
        Ok(CreateCourse {  
            tutor_id: new_course.tutor_id,
```

```

        course_name: new_course.course_name.clone(),
        course_description: new_course.course_description.clone(),
        course_format: new_course.course_format.clone(),
        course_structure: new_course.course_structure.clone(),
        course_level: new_course.course_level.clone(),
        course_duration: new_course.course_duration.clone(),
        course_language: new_course.course_language.clone(),
        course_price: new_course.course_price,
    })
}
}

```

Note that *Error* is a type placeholder associated with the *TryFrom* trait. We are declaring it to be of type *EzyTutorError* since we would like to unify all error handling with the *EzyTutorError* type. Within the function, we can then raise errors of type *EzyTutorError* in case of faults.

However for our purposes here, it would suffice to use the *From* trait, as we do not anticipate any failure conditions during this conversion. Usage of *TryFrom* trait is only shown here to demonstrate how to use it if the need arises.

We now have a way to receive data from an api client for creating a new course. What about course updates? Can we use the same *CreateCourse* struct? We cannot. This is because, while updating a course, we don't want to allow the *tutor_id* to be modified as we don't want the course created by one tutor to be switched over to another tutor. Secondly, the *course_name* field in *CreateCourse* struct is mandatory. For updating a course, we don't want to force the user to update the name everytime. So, let's create another struct that's more suitable for updating course details.

```

#[derive(Deserialize, Debug, Clone)]
pub struct UpdateCourse {
    pub course_name: Option<String>,
    pub course_description: Option<String>,
    pub course_format: Option<String>,
    pub course_structure: Option<String>,
    pub course_duration: Option<String>,
    pub course_price: Option<i32>,
    pub course_language: Option<String>,
    pub course_level: Option<String>,
}

```

Note that all the fields here are optional, which is the way it should be for a good user experience.

We'll also have to write the *From* trait implementation for *UpdateCourse*, similar to that for *CreateCourse*. Here is the code:

```
impl From<web::Json<UpdateCourse>> for UpdateCourse {
    fn from(update_course: web::Json<UpdateCourse>) -> Self {
        UpdateCourse {
            course_name: update_course.course_name.clone(),
            course_description: update_course.course_description,
            course_format: update_course.course_format.clone(),
            course_structure: update_course.course_structure.clone(),
            course_level: update_course.course_level.clone(),
            course_duration: update_course.course_duration.clone(),
            course_language: update_course.course_language.clone(),
            course_price: update_course.course_price,
        }
    }
}
```

Before we forget, in the file `$PROJECT_ROOT/src/iter5/models/course.rs`, delete the *From* trait implementation to convert from `web::Json<Course>` to `Course` struct, which we wrote in the previous chapter, as we now have separate structs for receiving data from users (*CreateCourse* and *UpdateCourse*) and for sending data back (`Course`).

This concludes the data model changes for the `Course` data struct.

However, we're not done yet. We have to change the model of the physical database tables to add the new fields.

In the `course.sql` file under `$PROJECT_ROOT/src/iter5/dbscripts`, add the following database scripts:

```
/* Drop tables if they already exist */

drop table if exists ezy_course_c6;

/* Create tables. */
/* Note: Don't put a comma after last field */

create table ezy_course_c6
```

```

(
    course_id serial primary key,
    tutor_id INT not null,
    course_name varchar(140) not null,
    course_description varchar(2000),
    course_format varchar(30),
    course_structure varchar(200),
    course_duration varchar(30),
    course_price INT,
    course_language varchar(30),
    course_level varchar(30),
    posted_time TIMESTAMP default now()
);

```

Note the main changes compared to the script we wrote in the previous chapter:

1. Database table name now has the c6 suffix. This is to allow us to test the code for each chapter independently.
2. The additional data elements we have designed in the *Course* data structure are reflected in the table creation script.
3. Note in particular the use of *NOT NULL* constraint specified for *tutor_id* and *course_name*. This will be enforced by the database, and we'll not be able to add a record without these columns. In addition, *course_id* which is marked as the primary key and *posted_time* which is automatically set to current time by default, are also enforced at the database-level. The rest of the fields that do not have a *NOT NULL* constraint are optional columns. If you refer back to the *Course* struct, you'll notice that these columns are also the ones marked as *Option<T>* type in the *Course* struct definition. In this way, we have aligned the database column constraints with the Rust struct.

To test the database script, run the following command from the command-line. Make sure the right path to the script file is specified.

```
psql -U <user-name> -d ezytutors < <path.to.file>/course.sql
```

Replace <user-name> and <path.to-file> with your own, and enter password if prompted. You should see the scripts execute successfully. To verify that the tables have indeed been created as per the script specification, login to

psql shell with the following command and verify:

```
psql -U <user-name> -d ezytutors
\d #1
\dp ezy_course_c6 #2
\q #3
```

After creating the new table, we need to give permissions to the database user. Run the following commands from the terminal command-line.

```
psql -U <user-name> -d ezytutors // Login to psql shell
GRANT ALL PRIVILEGES ON TABLE __ezy_course_c6__ to <user-name>
\q // Quit the psql shell
```

Replace the <user-name> with your own, and execute the commands. This <user-name> should be the same as that you've configured in the *.env* file. Note that you can also choose to execute this step directly as part of the database scripts after creating the table, should you choose.

With this, we conclude the data model changes.

In the next subsection, let's make the changes to the API processing logic to accommodate the data model changes.

6.2.2 Making changes to Course APIs

Recall that in the previous section, we enhanced the data model for *Course*, and created new database scripts to create the new structure of the *Course* that we have designed.

We'll now have to modify the application logic to incorporate the data model changes. To verify this, just run the following command from the *project root*:

```
cargo check
```

You'll see that there are errors in the database access and handler functions that need to be fixed. Let's do that now.

We'll start with the routes in *\$PROJECT_ROOT/src/iter5/routes.rs*. Modify

the code to look like this:

```
use crate::handlers::{course::*, general::*};
use actix_web::web;

pub fn general_routes(cfg: &mut web::ServiceConfig) {
    cfg.route("/health", web::get().to(health_check_handler));
}

pub fn course_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/courses")
            .route("", web::post().to(post_new_course))           #1
            .route("/{tutor_id}", web::get().to(get_courses_for_tutor))
            .route("/{tutor_id}/{course_id}", web::get().to(get_course_d
            .route(
                "/{tutor_id}/{course_id}",
                web::put().to(update_course_details),             #4
            )
            .route("/{tutor_id}/{course_id}", web::delete().to(delete_co
        );
    }
}
```

Note that we are importing the handler functions from two modules:
crate::handlers::course and *crate::handlers::general*.

Also note the use of the appropriate HTTP methods for the various routes, for example *post()* method to create a new course, *get()* method to retrieve a single course or a list of courses, *put()* method to update a course and *delete()* method for deleting a course.

Notice also the use of URL path parameters *{tutor_id}* and *{course_id}* to identify specific resources on which to operate.

You may have a question at this point about the *CreateCourse* and *UpdateCourse* structs that we designed as part of the data model to enable creation and update of course records. Why are they not visible in the routes definition? This is because these structs are sent as part of the HTTP request payload, which is automatically extracted by Actix and made available to the respective handler functions. Only the URL Path parameters, HTTP methods and names of handler functions for a route are specified as part of the routes declaration in Actix Web.

Let's next focus on the handler functions in
\$PROJECT_ROOT/src/iter5/handlers/course.rs.

Here are the module imports:

```
use crate::dbaccess::course::*;
use crate::errors::EzyTutorError;
use crate::models::course::{CreateCourse, UpdateCourse};
use crate::state::AppState;
use actix_web::{web, HttpResponse};
```

First, recall that the handler functions are called whenever an HTTP request arrives at one of the routes defined in *routes.rs*. In case of *courses*, for example, it can be a *GET* request to retrieve a list of courses for a tutor or a *POST* request to create a new course. The handler functions corresponding to each of the valid course routes will be stored in this file. The handler functions, in turn, make use of the *Course* data models and database access functions, and these are reflected in the module imports.

Here, we're importing the database access functions (as the handlers will invoke them), custom error type, data structures from the *Course* model, *AppState* (for database connection pool), and Actix utilities needed for HTTP communications with the client front-end.

Let's write the various handler functions corresponding to the routes, one by one.

Here is the handler method to retrieve all courses for a tutor.

```
pub async fn get_courses_for_tutor(
    app_state: web::Data<AppState>,
    path: web::Path<i32>,
) -> Result<HttpResponse, EzyTutorError> {
    let tutor_id = path.into_inner();
    get_courses_for_tutor_db(&app_state.db, tutor_id)
        .await
        .map(|courses| HttpResponse::Ok().json(courses))
}
```

This function accepts a URL path parameter that refers to the *tutor_id*, which is encapsulated in the Actix data structure `web::Path<i32>`. The function

returns an HTTP Response containing either the data requested, or an error message.

The handler function in turn invokes the database access function `get_courses_for_tutor_db` to access the database and retrieve the course list.

The return value from the database access function is handled through the `map` construct in Rust to construct a valid HTTP Response message with the success code, and send the list of courses back as part of the HTTP response body.

In case of errors while accessing the database, the database access functions raise error of type `EzyTutorError` which is then propagated back to the handler functions, where this error is transformed into an Actix error type, and then sent back to the client through a valid HTTP Response message. This error translation is handled by the Actix framework, provided the application implements the Actix `ResponserError` trait on the `EzyTutorError` type, which we have done in the previous chapter.

Let's next look at the code for retrieving an individual course record.

```
pub async fn get_course_details(
    app_state: web::Data<AppState>,
    path: web::Path<(i32, i32)>,
) -> Result<HttpResponse, EzyTutorError> {
    let (tutor_id, course_id) = path.into_inner();
    get_course_details_db(&app_state.db, tutor_id, course_id)
        .await
        .map(|course| HttpResponse::Ok().json(course))
}
```

Similar to the previous function, this function is also invoked in response to an `HTTP::GET` request. The difference is that here, we will receive the `tutor_id` and `course_id` as part of URL path parameters, which will help us uniquely identify a single course record in the database.

Note the use of `.await` keyword in these handler functions while invoking the corresponding database access functions. Since the database access library we use, `sqlx`, uses an asynchronous connection to the database, we use the `.await` keyword to denote an asynchronous call to communicate with the database.

Moving on, here is the code for the handler function to post a new course.

```
pub async fn post_new_course(
    new_course: web::Json<CreateCourse>,
    app_state: web::Data<AppState>,
) -> Result<HttpResponse, EzyTutorError> {
    post_new_course_db(&app_state.db, new_course.into()?)
        .await
        .map(|course| HttpResponse::Ok().json(course))
}
```

This handler function is invoked for an HTTP::POST request received on the route specified in the *routes.rs* file. The Actix framework deserializes the HTTP request body of this POST request, and makes the data available to the *post_new_course()* handler function within the *web::Json<CreateCourse>* data structure. Recall that we have written a conversion method to convert from *web::Json<CreateCourse>* to *CreateCourse* struct as part of the *From* trait implementation in *models/course.rs* file, which we are invoking within the handler function using the expression *new_course.into()?*. Note that if we had implemented the conversion function using the *TryFrom* trait instead of the *From* trait, we would invoke the conversion using *new_course.try_into()?*, with the ? denoting the possibility of an error being returned from the conversion function.

In this handler function, after a new course is created, the database access function returns the newly created course record, which is then sent back from the web service within the body of an HTTP Response message.

Next, let's look at the handler function to delete a course.

```
pub async fn delete_course(
    app_state: web::Data<AppState>,
    path: web::Path<(i32, i32)>,
) -> Result<HttpResponse, EzyTutorError> {
    let (tutor_id, course_id) = path.into_inner();
    delete_course_db(&app_state.db, tutor_id, course_id)
        .await
        .map(|resp| HttpResponse::Ok().json(resp))
}
```

This handler function is invoked in response to an HTTP::DELETE request. The handler function invokes the *delete_course_db* database access function

to perform the actual deletion of the course record in the database. On successful deletion, the handler function receives a message confirming successful deletion, which is then sent back as part of the HTTP Response.

Here is the handler function to update details for a course.

```
pub async fn update_course_details(
    app_state: web::Data<AppState>,
    update_course: web::Json<UpdateCourse>,
    path: web::Path<(i32, i32)>,
) -> Result<HttpResponse, EzyTutorError> {
    let (tutor_id, course_id) = path.into_inner();
    update_course_details_db(&app_state.db, tutor_id, course_id,
        .await
        .map(|course| HttpResponse::Ok().json(course))
}
```

This handler function is invoked in response to an HTTP::PUT request on the specified route in *routes.rs* file. It receives two URL path parameters - *tutor_id* and *course_id* which are used to uniquely identify a course in the database. The input parameters for the course to be modified are sent from the web/API front-end to the Actix web server route as part of the HTTP request body, and this is made available by Actix to the handler function as *web::Json::UpdateCourse*.

Note the use of *update_course.into()* expression. This is used to convert *web::json::UpdateCourse* to *UpdateCourse struct*. To achieve this, we've earlier implemented the *From* trait in the *models/course.rs* file.

The updated course details are then sent back as part of the HTTP response message.

Let's also write the unit test cases for the handler functions.

In the *handlers/course.rs* file, add test cases (after the code for handler functions) within the *tests* module as shown:

```
#[cfg(test)]
mod tests {
    //write test cases here
}
```

All the test cases and module imports for testing should be placed within this *tests* module block.

Let's add the module imports first:

```
use super::*;

use actix_web::http::StatusCode;
use actix_web::ResponseError;
use dotenv::dotenv;
use sqlx::postgres::PgPool;
use std::env;
use std::sync::Mutex;
```

Let's start with the test case for getting all courses for a tutor.

```
#[actix_rt::test]
async fn get_all_courses_success() {
    dotenv().ok(); #1
    let database_url = env::var("DATABASE_URL").expect("DATAB");
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    );
    let tutor_id: web::Path<i32> = web::Path::from(1);      #
    let resp = get_courses_for_tutor(app_state, tutor_id).awa
    assert_eq!(resp.status(), StatusCode::OK);               #
}
```

Here is the test case to retrieve an individual course.

```
#[actix_rt::test]
async fn get_course_detail_success_test() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB");
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    );
    let parameters: web::Path<(i32, i32)> = web::Path::from((
        let resp = get_course_details(app_state, parameters).awa
        assert_eq!(resp.status(), StatusCode::OK);
```

```
}
```

The test function is mostly similar to the previous one, except that here we are retrieving a single course from the database.

What happens if we provide an invalid *course id* or *tutor id*? In the handler and database access functions, we handle such a case by returning an error. Let's see if we can verify this scenario.

```
#[actix_rt::test]
async fn get_course_detail_failure_test() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB");
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    );
    let parameters: web::Path<(i32, i32)> = web::Path::from((
        let resp = get_course_details(app_state, parameters).awa
        match resp {
            Ok(_) => println!("Something wrong"),
            Err(err) => assert_eq!(err.status_code(), StatusCode:
        }
    )
}
```

Next, we'll write the test case to post a new course.

```
#[ignore]
#[actix_rt::test]
async fn post_course_success() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB");
    let pool: PgPool = PgPool::new(&database_url).await.unwra
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    );
    let new_course_msg = CreateCourse { #1
        tutor_id: 1,
        course_name: "Third course".into(),
        course_description: Some("This is a test course".into(
        course_format: None,
```

```

        course_level: Some("Beginner".into()),
        course_price: None,
        course_duration: None,
        course_language: Some("English".into()),
        course_structure: None,
    };
    let course_param = web::Json(new_course_msg); #2
    let resp = post_new_course(course_param, app_state).await
    assert_eq!(resp.status(), StatusCode::OK);
}

```

Rest of the code is largely similar to the previous test cases. Note also the usage of `#[ignore]` at the top of the test case. This ensures that `cargo test` command will ignore this test case whenever it is invoked. This is because we may not want to create a new test case everytime we run test cases for sanity checks, and in such a case, we can use the `#[ignore]` annotation.

Shown next is the test case to update a course.

```

#[actix_rt::test]
async fn update_course_success() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    });
    let update_course_msg = UpdateCourse { #1
        course_name: Some("Course name changed".into()),
        course_description: Some("This is yet another test co
        course_format: None,
        course_level: Some("Intermediate".into()),
        course_price: None,
        course_duration: None,
        course_language: Some("German".into()),
        course_structure: None,
    };
    let parameters: web::Path<(i32, i32)> = web::Path::from((
    let update_param = web::Json(update_course_msg);
    let resp = update_course_details(app_state, update_param,
        .await
        .unwrap();
    assert_eq!(resp.status(), StatusCode::OK);
}

```

Lastly, here is the test case to delete a course.

```
#[ignore]
#[actix_rt::test]
async fn delete_test_success() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB");
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    );
    let parameters: web::Path<(i32, i32)> = web::Path::from((
        let resp = delete_course(app_state, parameters).await.unw
        assert_eq!(resp.status(), StatusCode::OK);
    }
}
```

What if we were to provide an invalid *tutor-id* or *course-id*? Let's write a test case for that.

```
#[actix_rt::test]
async fn delete_test_failure() {
    dotenv().ok();
    let database_url = env::var("DATABASE_URL").expect("DATAB");
    let pool: PgPool = PgPool::connect(&database_url).await.u
    let app_state: web::Data<AppState> = web::Data::new(AppSt
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        db: pool,
    );
    let parameters: web::Path<(i32, i32)> = web::Path::from((
        let resp = delete_course(app_state, parameters).await;
        match resp {
            Ok(_) => println!("Something wrong"),
            Err(err) => assert_eq!(err.status_code(), StatusCode:
        }
    }
}
```

This concludes the unit test cases for the various handler functions. But we're not yet ready to run the tests as we have not implemented the database access functions. Let's look at them now in
\$PROJECT_ROOT/src/iter5/dbaccess/course.rs.

Let's begin with the database access function to retrieve all courses for a tutor, along with all the module imports for the file.

```
use crate::errors::EzyTutorError;
use crate::models::course::*;
use sqlx::postgres::PgPool;

pub async fn get_courses_for_tutor_db(
    pool: &PgPool,
    tutor_id: i32,
) -> Result<Vec<Course>, EzyTutorError> {
    // Prepare SQL statement

    let course_rows: Vec<Course> = sqlx::query_as!(<# Course,
        "SELECT * FROM ezy_course_c6 where tutor_id = $1",
        tutor_id
    )
    .fetch_all(pool) <# .await?; <#
    Ok(course_rows) <#
}
```

The `query_as!` macro comes in handy to map the columns in the database record into the `Course` data struct. This mapping is done automatically by `sqlx` if the `sqlx::FromRow` trait is implemented for `Course` struct. We have done this in the `models` module by auto-deriving this trait as outlined here.

```
#[derive(Deserialize, Serialize, Debug, Clone, sqlx::FromRow)]
pub struct Course {
    // fields
}
```

Without the `query_as!` macro, we would have to manually perform the mapping of each database column to the corresponding `Course` struct field.

Here is the next function to retrieve a single course from the database.

```
pub async fn get_course_details_db(
    pool: &PgPool,
    tutor_id: i32,
    course_id: i32,
```

```

) -> Result<Course, EzyTutorError> {
    // Prepare SQL statement
    let course_row = sqlx::query_as!(
        Course,
        "SELECT * FROM ezy_course_c6 where tutor_id = $1 and course
        tutor_id,
        course_id
    )
    .fetch_optional(pool)                      #2
    .await?;

    if let Some(course) = course_row {          #3
        Ok(course)
    } else {                                    #4
        Err(EzyTutorError::NotFound("Course id not found".into()))
    }
}

```

The code for adding a new course to the database is shown here:

```

pub async fn post_new_course_db(
    pool: &PgPool,
    new_course: CreateCourse,
) -> Result<Course, EzyTutorError> {
    let course_row = sqlx::query_as!(Course, "insert into ezy_courses
        new_course.tutor_id, new_course.course_name, new_course.course_duration,
        new_course.course_level, new_course.course_description
    .fetch_one(pool)                      #2
    .await?;

    Ok(course_row)
}

```

Specifically, note the usage of the *returning* keyword in the sql insert statement. This is a feature supported by Postgres database, which enables us to retrieve the newly inserted course details as part of the same insert query (instead of having to write a separate sql query).

Let's look at the function to delete a course from the database.

```

pub async fn delete_course_db(
    pool: &PgPool,
    tutor_id: i32,
    course_id: i32,
) -> Result<String, EzyTutorError> {

```

```

// Prepare SQL statement
let course_row = sqlx::query!(
    "DELETE FROM ezy_course_c6 where tutor_id = $1 and course_
    tutor_id,
    course_id,
)
.execute(pool)                                #2
.await?;
Ok(format!("Deleted {:#?} record", course_row))   #3
}

```

Lastly, let's look at the code to update details of a course.

```

pub async fn update_course_details_db(
    pool: &PgPool,
    tutor_id: i32,
    course_id: i32,
    update_course: UpdateCourse,
) -> Result<Course, EzyTutorError> {
    // Retrieve current record

    let current_course_row = sqlx::query_as!(                #1
        Course,
        "SELECT * FROM ezy_course_c6 where tutor_id = $1 and cour
        tutor_id,
        course_id
    )
    .fetch_one(pool)                                     #2
    .await
    .map_err(|_err| EzyTutorError::NotFound("Course id not found"))

    // Construct the parameters for update:                      #4
    let name: String = if let Some(name) = update_course.course_n
        name
    } else {
        current_course_row.course_name
    };
    let description: String = if let Some(desc) = ... #9
    let format: String = if let Some(format) = ... #9
    let structure: String = if let Some(structure) = ... #9
    let duration: String = if let Some(duration) = ... #9
    let level: String = if let Some(level) = ... #9
    let language: String = if let Some(language) = ... #9
    let price = if let Some(price) = ... #9

    // Prepare SQL statement

```

```

let course_row =
    sqlx::query_as!(
        Course,
        "UPDATE ezy_course_c6 set course_name = $1, course_descri
course_structure = $4, course_duration = $5, course_price
course_level = $8 where tutor_id = $9 and course_id = $10
course_name, course_description, course_duration, course_
course_language, course_structure, course_price, posted_t
structure, duration, price, language, level, tutor_id, cou
)
    .fetch_one(pool)                                #6
    .await;
if let Ok(course) = course_row {                  #7
    Ok(course)
} else {
    Err(EzyTutorError::NotFound("Course id not found".into()))
}
}

```

Note the lines of code corresponding to coding annotation <4>. Since the *UpdateCourse* struct contains a set of optional fields, we will have to first verify which field has been sent by the api client. If a new value has been sent for a field, we need to update it. Otherwise, we need to retain the original value present in the database. To achieve this, we are first extracting the current course record containing all the fields. Then if the value of a particular field is sent by the api client, we use it to update the database, otherwise we use the existing value to update.

With this, we've now completed the code changes to the data model, routes, handlers, test cases, and database access functions for courses.

You can now check for any compilation errors by running this command from the \$PROJECT_ROOT:

```
cargo check
```

If it compiles successfully, you can build and run the server with:

```
cargo run
```

You can test the HTTP::GET related APIs from the browser with:

```
http://localhost:3000/courses/1      #1
```

```
http://localhost:3000/courses/1/2      #2
```

The POST, PUT and DELETE APIs can be tested with Curl or from a GUI tool such as *Postman*. The *Curl* commands shown here can be executed on the command-line from \$PROJECT_ROOT.

```
curl -X POST localhost:3000/courses -H "Content-Type: application  
-d '{"tutor_id":1, "course_name":"This is a culinary course", "c  
curl -X PUT localhost:3000/courses/1/5 -H "Content-Type: applicat  
curl -X DELETE http://localhost:3000/courses/1/6      #3
```

Ensure to change the *course_id* and *tutor_id* values based on your database data setup.

Further, you can run the test cases with:

```
cargo test
```

You can selectively disable the tests to be ignored with the #[ignore] annotation at the beginning of a test case function declaration.

With this, we come to the end of the changes for the *Course* related functionality. We've covered a lot of ground, which is summarized here:

1. We made changes to the *Course* data model to add additional fields, some of which are also optional values requiring the use of *Option<>* type in struct member declaration
2. We added data structures for creating and updating a course
3. We implemented conversion methods from Actix Json data structs into *CreateCourse* and *UpdateCourse* structs. We saw how to use both *TryFrom* and *From* traits
4. We modified the routes to cover create, update, delete and retrieve functions for course data.
5. We wrote the handler functions for each of these routes.
6. We wrote the unit test cases for each handler function. We wrote a couple of test cases where errors are returned from the handler functions, instead of a success response.
7. We wrote the database access functions corresponding to each handler

method. We saw the usage of `query_as!` macro to significantly reduce the boilerplate code for mapping columns from a database record into Rust struct fields.

Are you exhausted already? Writing real-world web services and applications involve considerable work for sure.

In the next section, we'll add functionality for data maintenance of tutors.

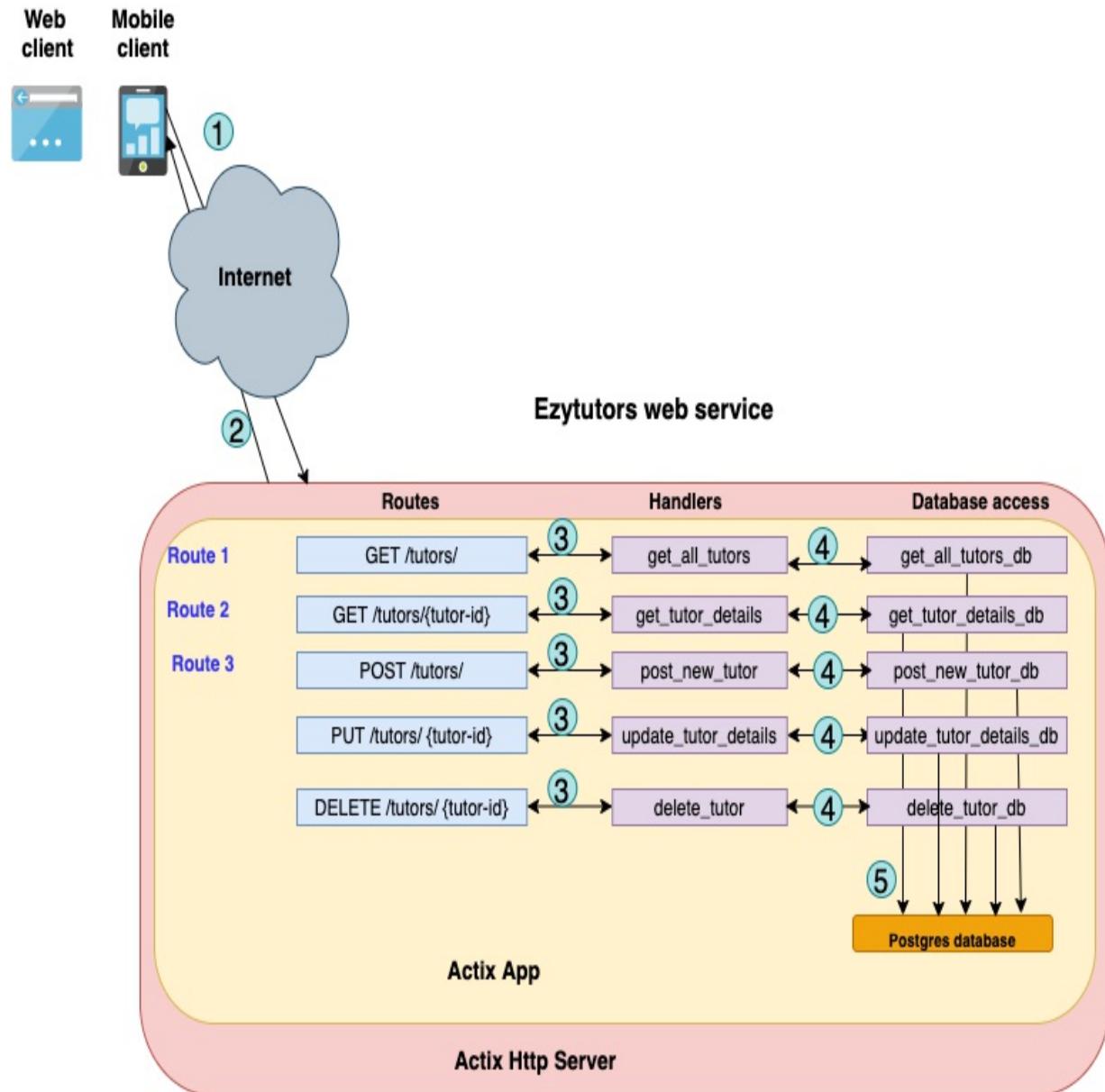
6.3 Enabling tutor registration and management

In this section, we'll design and write the code for tutor-related APIs which will include the Rust data models for tutors, database table structure, routes, handlers and database access functions for tutor data management.

Figure 6.3 shows the overall code structure for tutor-related APIs.

Figure 6.3. Code structure for tutor-related APIs

Tutor-related APIs



You'll notice that we have five routes. There is also a handler function and a database access function corresponding to each route.

Let's first look at the data model and routes.

6.3.1 Data model and routes for tutor

Let's first add a new struct *Tutor* to the data model, in

`$PROJECT_ROOT/src/iter5/models/tutor.rs` file.

Start with the module imports.

```
use actix_web::web;
use serde::{Deserialize, Serialize};
```

Define the struct as shown.

```
#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct Tutor {
    tutor_id: i32,
    tutor_name: String,
    tutor_pic_url: String,
    tutor_profile: String
}
```

We've defined a *Tutor* struct that contains the following information:

- *Tutor Id*: this would be a unique id to represent a tutor, and will be auto-generated by the database.
- *Tutor name*: Full name of the tutor.
- *Tutor picture URL*: URL of the tutor photo/image.
- *Tutor profile*: A brief profile of the tutor.

Let's create two more structs, one to define the fields needed to create a new course, and another for updating it.

```
#[derive(Deserialize, Debug, Clone)]
pub struct NewTutor {
    pub tutor_name: String,
    pub tutor_pic_url: String,
    pub tutor_profile: String,
}
#[derive(Deserialize, Debug, Clone)]
pub struct UpdateTutor {
    pub tutor_name: Option<String>,
    pub tutor_pic_url: Option<String>,
    pub tutor_profile: Option<String>,
}
```

We need two separate structs because for creating a tutor, we require all the fields, but for updating, all fields are optional.

And similar to what we did for the *Course* data struct, here are the functions to convert from `web::Json<NewTutor>` to *NewTutor*, and `web::Json<UpdateTutor>` to *UpdateTutor*.

```
impl From<web::Json<NewTutor>> for NewTutor {
    fn from(new_tutor: web::Json<NewTutor>) -> Self {
        NewTutor {
            tutor_name: new_tutor.tutor_name.clone(),
            tutor_pic_url: new_tutor.tutor_pic_url.clone(),
            tutor_profile: new_tutor.tutor_profile.clone(),
        }
    }
}

impl From<web::Json<UpdateTutor>> for UpdateTutor {
    fn from(new_tutor: web::Json<UpdateTutor>) -> Self {
        UpdateTutor {
            tutor_name: new_tutor.tutor_name.clone(),
            tutor_pic_url: new_tutor.tutor_pic_url.clone(),
            tutor_profile: new_tutor.tutor_profile.clone(),
        }
    }
}
```

This completes the data model changes for *Tutor*.

Next, let's add the tutor-related routes in
`$PROJECT_ROOT/src/iter5/routes.rs`.

```
pub fn tutor_routes(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::scope("/tutors")
            .route("/", web::post().to(post_new_tutor))          #
            .route("/", web::get().to(get_all_tutors))          #
            .route("/{tutor_id}", web::get().to(get_tutor_details))
            .route("/{tutor_id}", web::put().to(update_tutor_data))
            .route("/{tutor_id}", web::delete().to(delete_tutor))
    );
}
```

Don't forget to update the module imports, to import the handler functions for tutors, which we will shortly write, in file
`$PROJECT_ROOT/src/iter5/routes.rs`.

```
use crate::handlers::{course::*, general::*, tutor::*};
```

We'll have to register the new tutor-routes in the main() function. Otherwise, the Actix framework will not recognize requests coming on the tutor routes, and also will not know how to route them to their handlers.

In `$PROJECT_ROOT/src/bin/iter5.rs`, add tutor routes after course routes while constructing the Actix App as shown here:

```
.configure(course_routes)
.configure(tutor_routes)
```

We can now move on to the handler functions in the next section.

6.3.2 Handler functions for tutor routes

We've already seen how handler functions are written for *Course*. Let's move quickly, only slowing down to look at any differences.

Here is the first handler method to retrieve all tutors, along with the module imports. Add this code to the `$PROJECT_ROOT/src/iter5/handlers/tutor.rs` file.

```
use crate::dbaccess::tutor::*;
use crate::errors::EzyTutorError;
use crate::models::tutor::{NewTutor, UpdateTutor};
use crate::state::AppState;

use actix_web::{web, HttpResponse};

pub async fn get_all_tutors(app_state: web::Data<AppState>) -> Re
    get_all_tutors_db(&app_state.db)
        .await
        .map(|tutors| HttpResponse::Ok().json(tutors))
}

pub async fn get_tutor_details(
    app_state: web::Data<AppState>,
    web::Path(tutor_id): web::Path<i32>,
) -> Result<HttpResponse, EzyTutorError> {
    get_tutor_details_db(&app_state.db, tutor_id)
        .await
        .map(|tutor| HttpResponse::Ok().json(tutor))
}
```

```
}
```

The two functions linked to HTTP::GET request are shown here. *get_all_tutors()* takes no parameters, while *get_tutor_details()* takes a *tutor_id* as a path parameter. Both invoke database access functions with the same name as the handler functions, but with a *db* suffix. The return value from the database access function is returned back to the web client in the body of an *HttpResponse* message.

Here are the handler functions for posting a new tutor entry, updating tutor details and deleting a tutor from the database.

```
pub async fn post_new_tutor(
    new_tutor: web::Json<NewTutor>,
    app_state: web::Data<AppState>,
) -> Result<HttpResponse, EzyTutorError> {
    post_new_tutor_db(&app_state.db, NewTutor::from(new_tutor))
        .await
        .map(|tutor| HttpResponse::Ok().json(tutor))
}

pub async fn update_tutor_details(
    app_state: web::Data<AppState>,
    web::Path(tutor_id): web::Path<i32>,
    update_tutor: web::Json<UpdateTutor>,
) -> Result<HttpResponse, EzyTutorError> {
    update_tutor_details_db(&app_state.db, tutor_id, UpdateTutor{
        .await
        .map(|tutor| HttpResponse::Ok().json(tutor))
}

pub async fn delete_tutor(
    app_state: web::Data<AppState>,
    web::Path(tutor_id): web::Path<i32>,
) -> Result<HttpResponse, EzyTutorError> {
    delete_tutor_db(&app_state.db, tutor_id)
        .await
        .map(|tutor| HttpResponse::Ok().json(tutor))
}
```

Here we see the three functions that are similar to how we did it for courses. The functional syntax of Rust makes the code really crisp and pleasant to read.

As an exercise, you can write the test cases for these handler methods. Refer back to the test cases for courses, if you have any doubts. In any case, the test cases will be available as part of the Git repo for the chapter, and are not included here only to restrict the length of this chapter. In the next section, we'll address the database access layer.

6.3.3 Database access functions for tutor routes

We'll now look at the database access functions for tutors. These should be placed in `$PROJECT_ROOT/src/iter5/dbaccess/tutor.rs` file.

Here is the database access function to get the list of tutors, along with the module imports:

```
use crate::errors::EzyTutorError;
use crate::models::tutor::{NewTutor, Tutor, UpdateTutor};
use sqlx::postgres::PgPool;

pub async fn get_all_tutors_db(pool: &PgPool) -> Result<Vec<Tutor> {
    // Prepare SQL statement
    let tutor_rows =
        sqlx::query!("SELECT tutor_id, tutor_name, tutor_pic_url,
                      .fetch_all(pool)
                      .await?";
    // Extract result

    let tutors: Vec<Tutor> = tutor_rows
        .iter()
        .map(|tutor_row| Tutor {
            tutor_id: tutor_row.tutor_id,
            tutor_name: tutor_row.tutor_name.clone(),
            tutor_pic_url: tutor_row.tutor_pic_url.clone(),
            tutor_profile: tutor_row.tutor_profile.clone(),
        })
        .collect();
    match tutors.len() {
        0 => Err(EzyTutorError::NotFound("No tutors found".into()))
        _ => Ok(tutors),
    }
}
```

Note that we're not using the `query_as!` macro to map the retrieved database records into `Tutor` struct. Instead, we are manually performing this mapping

within the *map* method. You may wonder why we have to take a more tedious approach compared to having the mapping automatically done by *sqlx* using *query_as!* macro. There are two main reasons for this:

1. The *query_as!* macros works as long as the field names in struct match the database column names. But there may be situations where this may not be feasible
2. Secondly, you may have additional fields in the struct compared to database columns. For example you may want to have a derived/computed field, or you may want a Rust struct to represent a tutor along with the list of her courses. In such cases, it is necessary to know how to perform this database-to-struct mapping manually. Hence, we are taking this approach more as a learning exercise, as it is always useful to have a wider repertoire of tools in a developer's kit.

Here is the database function to retrieve details for an individual tutor:

```
pub async fn get_tutor_details_db(pool: &PgPool, tutor_id: i32) ->
    // Prepare SQL statement
    let tutor_row = sqlx::query!(
        "SELECT tutor_id, tutor_name, tutor_pic_url, tutor_profile
         WHERE tutor_id = ?"
    )
    .bind(tutor_id)
    .fetch_one(pool)
    .await
    .map(|tutor_row| {
        Tutor {
            tutor_id: tutor_row.tutor_id,
            tutor_name: tutor_row.tutor_name,
            tutor_pic_url: tutor_row.tutor_pic_url,
            tutor_profile: tutor_row.tutor_profile,
        }
    })
    .map_err(|_err| EzyTutorError::NotFound("Tutor id not found"));

    Ok(tutor_row)
}
```

Note the particular use of *map_err* here. If there is no record found in the database, a *sqlx* error is returned, which we are converting to an *EzyTutorError* type using *map_err*, before propagating the error back to the

calling handler function using ? operator.

Here is the function to post a new tutor:

```
pub async fn post_new_tutor_db(pool: &PgPool, new_tutor: NewTutor
    let tutor_row = sqlx::query!("insert into ezy_tutor_c6 (tutor
        .fetch_one(pool)
        .await?;
        //Retrieve result
        Ok(Tutor {
            tutor_id: tutor_row.tutor_id,
            tutor_name: tutor_row.tutor_name,
            tutor_pic_url: tutor_row.tutor_pic_url,
            tutor_profile: tutor_row.tutor_profile,
        })
    }
```

We're constructing a query to insert a new tutor record in *easy_tutor_c6* table. Then we're fetching the inserted row and mapping it to the Rust tutor struct, which is returned back to the handler function.

The code for updating and deleting a tutor is not shown here. I would suggest you write it as an exercise. The completed code will be available in the code repo for this chapter, which you can refer to in case of doubts.

6.3.4 Database scripts for tutor

We're done with the application logic for the APIs. But lastly, we'll have to create a new table in the database for tutors, before we can even compile this code (Note that sqlx performs compile-time checking of database table names, and columns, so the compilation will fail if any of these don't exist or if table description does not match the sql statements).

Place the following database script under
\$PROJECT_ROOT/src/iter5/dbscripts/tutor-course.sql.

```
/* Drop tables if they already exist*/
drop table if exists ezy_course_c6 cascade;          #1
drop table if exists ezy_tutor_c6;

/* Create tables. */
```

```

create table ezy_tutor_c6 (                                #2
    tutor_id serial primary key,
    tutor_name varchar(200) not null,
    tutor_pic_url varchar(200) not null,
    tutor_profile varchar(2000) not null
);

create table ezy_course_c6                               #3
(
    course_id serial primary key,
    tutor_id INT not null,
    course_name varchar(140) not null,
    course_description varchar(2000),
    course_format varchar(30),
    course_structure varchar(200),
    course_duration varchar(30),
    course_price INT,
    course_language varchar(30),
    course_level varchar(30),
    posted_time TIMESTAMP default now(),
    CONSTRAINT fk_tutor                         #4
    FOREIGN KEY(tutor_id)                      #4
        REFERENCES ezy_tutor_c6(tutor_id)      #4
        ON DELETE cascade
);

grant all privileges on table ezy_tutor_c6 to <username>;  #5
grant all privileges on table ezy_course_c6 to <username>;
grant all privileges on all sequences in schema public to <username>

/* Load seed data for testing */    #6
insert into ezy_tutor_c6(tutor_id, tutor_name, tutor_pic_url,tutor_profile)
values(1,'Merlene','http://s3.amazonaws.com/pic1', 'Merlene is a tutor');

insert into ezy_tutor_c6(tutor_id, tutor_name, tutor_pic_url,tutor_profile)
values(2,'Frank','http://s3.amazonaws.com/pic2', 'Frank is an experienced tutor');

insert into ezy_course_c6
    (course_id,tutor_id, course_name, course_level, posted_time)
values(1, 1, 'First course', 'Beginner' , '2021-04-12 05:40:00');
insert into ezy_course_c6
    (course_id, tutor_id, course_name, course_format, posted_time)
values(2, 1, 'Second course', 'ebook', '2021-04-12 05:45:00');

```

6.3.5 Run and test the tutor APIs

Run the following command from the command-line to execute the database script.

```
psql -U <user-name> -d ezytutors < <path.to.file>/tutor-course.sql
```

Replace <user-name> and <path.to-file> with your own, and enter password when prompted. You should see the scripts execute successfully. To verify that the tables have indeed been created as per the script specification, login to *psql* shell with the following command and verify:

```
psql -U <user-name> -d ezytutors
\d
\dt+ ezy_tutor_c6    #1
\dt+ ezy_course_c6   #2
\q                  #3
```

Compile the program to check for errors. After resolving any errors, build and run the web server with:

```
cargo check
cargo run --bin iter5
```

You can first run the automated tests with:

```
cargo test
```

Before running the test scripts ensure the data being queried for in test cases is present in the database, or prepare the data appropriately.

You can also manually execute the CRUD APIs from Curl for tutor as shown:

```
curl -X POST localhost:3000/tutors/ -H "Content-Type: application/json"
curl -X PUT localhost:3000/tutors/8 -H "Content-Type: application/json"
curl -X DELETE http://localhost:3000/tutors/8    #3
```

From a browser, you can execute the HTTP::GET apis with:

```
http://localhost:3000/tutors/           #1
http://localhost:3000/tutors/2          #2
```

As an exercise, you can also try deleting a tutor for which course records exist. You should receive an error message. This is because courses and tutors are linked by foreign-key constraint in the database. Once you delete all courses for a *tutor-id*, that tutor can be deleted from the database.

Another exercise you can try out is to provide an invalid json as part of creating or updating a tutor or course (for example by removing a double quote or a curly brace from json data for creating or updating a tutor). You'll find that neither does the command get executed on the server, nor do you get any error message stating that json is invalid. This is not user-friendly. To fix this, let's make a few changes.

In the file *ezytutors/tutor-db/src/iter5/errors.rs*, add a new entry *InvalidInput(String)* in the *EzyTutorError* enum, which will then look like this:

```
#[derive(Debug, Serialize)]
pub enum EzyTutorError {
    DBError(String),
    ActixError(String),
    NotFound(String),
    InvalidInput(String),
}
```

InvalidInput(String) denotes that *EzyTutorError* enum can take a new invariant - *InvalidInput*, that in turn can accept a *String* value as parameter. For all errors arising out of invalid parameters sent by the API client, we'll use this new variant.

Also in the same *errors.rs* file, make the following additional changes, caused by the addition of the new enum variant.

In the function *error_response()* add the code to deal with the *EzyTutorError::InvalidInput* type:

```
fn error_response(&self) -> String {
    match self {
        EzyTutorError::DBError(msg) => {
            println!("Database error occurred: {:?}", msg);
            "Database error".into()
        }
    }
}
```

```

        EzyTutorError::ActixError(msg) => {
            println!("Server error occurred: {:?}", msg);
            "Internal server error".into()
        }
        EzyTutorError::NotFound(msg) => {
            println!("Not found error occurred: {:?}", msg);
            msg.into()
        }
        EzyTutorError::InvalidInput(msg) => {
            println!("Invalid parameters received: {:?}", msg);
            msg.into()
        }
    }
}

```

In the *ResponseError* trait implementation, add code to deal with the new enum variant.

```

fn status_code(&self) -> StatusCode {
    match self {
        EzyTutorError::DBError(_msg) | EzyTutorError::ActixError(_m
            StatusCode::INTERNAL_SERVER_ERROR
        }
        EzyTutorError::InvalidInput(_msg) => StatusCode::BAD_REQUEST
        EzyTutorError::NotFound(_msg) => StatusCode::NOT_FOUND,
    }
}

```

We're now ready to make use of this new error variant in our code. Add the following code in `$PROJECT_ROOT/src/bin/iter5.rs`, while creating an Actix app instance, to raise an error if the Json data received at the server is invalid:

```

let app = move || {
    App::new()
        .app_data(shared_data.clone())
        .app_data(web::JsonConfig::default().error_handler(|_err, _re
            EzyTutorError::InvalidInput("Please provide valid Json input
        )))
        .configure(general_routes)
        .configure(course_routes)
        .configure(tutor_routes)
};

```

Now whenever you provide an invalid json data, you'll receive the specified

error message.

With this, we conclude this chapter that shows how to refactor code in Rust and Actix-web , and add functionality in a way that you as the developer retain complete control over the entire process. Our *tutor* web service is now more complex and aligned to the real-world, rather than being just an academic example. It has two types of entities (tutors and courses) that have a defined relationship between them at the database-level, and eleven API endpoints. It can handle five broad classes of errors - database-related errors, Actix-related errors, bad user input parameters, handling requests on resources that do not exist (Not_found), and badly-formatted json in input requests. It can seamlessly process concurrent requests as it uses async calls both in the actix-layer and database-access layer without any bottlenecks. The project code is well organized which will enable further evolution of the web service over time, and more importantly, will be easily understandable as newer developers take charge of the existing code base. The project code and configuration are separated by using the .env file which contains database access credentials and other such config information. Dependency injection is built into the project through Application state (in *state.rs*), which serves as a placeholder in which to add more dependencies that need to be propagated to the various handler functions. The project itself does not use too many external crates and eschews short-cuts and magical crates (such as crates that automate code generation for error handling or database functions), but the reader is encouraged to experiment with other third-party crates, with this foundational knowledge of doing things the hard way.

You'll observe that throughout this process, the Rust compiler has been a great friend and guide to help you achieve your goals.

If you have been able to follow me successfully until this step, I applaud your perseverance.

I hope this chapter has given you the confidence to fearlessly take on tasks to enhance any Rust web codebase, even if you were not the original author of the code.

6.4 Summary

- In this chapter, we enhanced the data model for courses, added more course API routes and evolved the code for handlers and database access along with the test cases.
- We also added functionality to allow creation, update, deletion and querying of tutor records. We created the database model and scripts to store tutor data, and defined the relationship between tutors and courses with foreign-key constraints. We created new routes for tutor-related CRUD APIs, wrote the handler functions, database access code and test cases.
- In the handler code, we saw how to create separate data structures for creation and update of tutor and course data, and how to use *From* and *TryFrom* traits to write functions for converting between data types. We also saw how to mark fields in data structures as *optional* using the *Option<T>* type, and map it to the corresponding column definitions in the database.
- In the database code, we learnt how to use *query_as!* macro to simplify and reduce boiler-plate code by auto-deriving *sqlx::FromRow* for the *Course* struct, where the mapping between database columns and fields of the *Course* struct was derived automatically by *sqlx*. We also learnt how to perform this mapping from database record to Rust structs manually, in cases where usage of *query_as!* macro is not possible or desirable.
- We learnt to write code in the handler and database access layers in a concise but highly readable manner using Rust's functional constructs.
- We strengthened knowledge of error-handling concepts by revisiting the entire error management workflow, and fine-tuning error handling to make the user experience more interactive and meaningful.
- We restructured the project code organisation to support projects as they get larger and more complex with separate and clearly marked areas to store code for handlers, database access functions, data models and database scripts. We also separated the source files that contain tutor and course functionality by organizing them into Rust modules.
- We saw how to test code using automated test scripts that can automatically handle both success and error conditions. We also tested the API scenarios using both Curl commands and from the browser.

For such refactoring, there isn't a specific order of steps that can be

prescribed, but generally it helps to start from the outside (user interface), and work your way through the various layers of the application. For example, if there is some new information requested from the web service, start with defining the new route, define the handler function, then the data model and database access function. If this necessitates changes to the database schema, modify the database creation/update scripts and also any associated migration scripts. The database access functions provide a layer of abstraction to switch to a different database, if needed, as part of refactoring. While the Rust compiler is your best friend to help you succeed in refactoring, your next best friend would be the automated test scripts that you wrote previously, that will help to ensure there is no regression of functionality.

With this, we also conclude the first part of the book, which is on developing a web service using Rust. We will however revisit a few more topics on the web service in the last portion of the book when we discuss how to prepare the web service and application for production deployment.

In the next part of the book, we'll move on to client-side web with Rust where we'll cover how to develop server-rendered web front-ends using Rust and Actix-web.

See you in the next chapter.

7 Introduction to server-side web apps in Rust

This chapter covers

- Serving a static web page with Actix
- Rendering a dynamic web page with Actix and Tera
- Adding user input with forms
- Displaying a list with templates
- Writing and running client-side tests
- Connecting to the backend web service

In *chapters 3-6* of the book, we built out the *Tutors web service* from scratch using Rust and the Actix web framework. In this section, we'll focus on learning the basics of building a web application in Rust.

It may sound strange that a system programming language is being used to create a web application. But that's the power of Rust. It can straddle the worlds of system and application programming with ease.

In this chapter, you will get introduced to concepts and tools for working with Rust to build web applications. At this point it is important to recall that there are two broad techniques for building web applications - *server-side rendering* (SSR) and *single page applications* (SPA), each possibly in the form of *progressive web application* (PWA). In this section, we'll focus on the former, and in later chapters we'll cover the latter. We will not cover PWAs in this book.

More specifically, the focus for *chapters 7-9* is to learn how develop a simple web application that can be used by users to register and login to a web application, view lists and detail views, and perform standard CRUD (create-read-update-delete) operations on data using web-based forms. Along the way, you will learn how to render dynamic web pages using the Actix web framework along with a template engine. While we can use any Rust web

framework (*Actix web*, *Rocket* and *Warp* to name a few) to achieve the same goal, staying with *Actix web* helps us leverage the learnings from the previous chapters.

With this background, we are ready to get started.

Server-side rendering is a web development technique where web pages are rendered on the server and then sent to the client (web browser). In this approach, a web application running on the server combines static HTML pages (e.g., from a web designer) with data (fetched either from a database or from other web services) and sends a fully-rendered web page to the browser for displaying to the user. Web applications that use such a technique are called *server-rendered* or *server-side* web apps. With this approach, websites load faster, and the web page content reflects that latest data, as every request typically involves fetching the latest copy of user data (exception is when caching techniques are adopted on the server). As a side note, to keep data specific for a user, web sites either require users to login to authenticate / identify themselves, or use cookies to personalize content for a user.

Web pages can either be *static* or *dynamic*.

An example of a *static* web page is the home screen of your bank website which typically serves as a marketing tool for the bank, and also provides useful links for its customers to use the services of the bank. This page is the same for whoever accesses the bank's home page URL. In this sense, it is a *static* web page.

A *dynamic* web page is what you see when you log in to your bank with your authorized credentials (such as a username and password), and view your account balances and statements. This page is dynamic in the sense that each customer views his or her own balance, but the web page may also contain static components such as the bank's logo and other common styling of the web page (such as colours, fonts, layout etc) which are shown to all customers viewing the account balances.

We know how to create a static web page. A web designer can do this either writing the HTML and CSS scripts by hand or use one of the many available tools for this purpose. But how does one convert a *static* web page to a

dynamic web page?

This is where a *template engine* comes in.

Figure 7.1. Server-side rendering of web pages

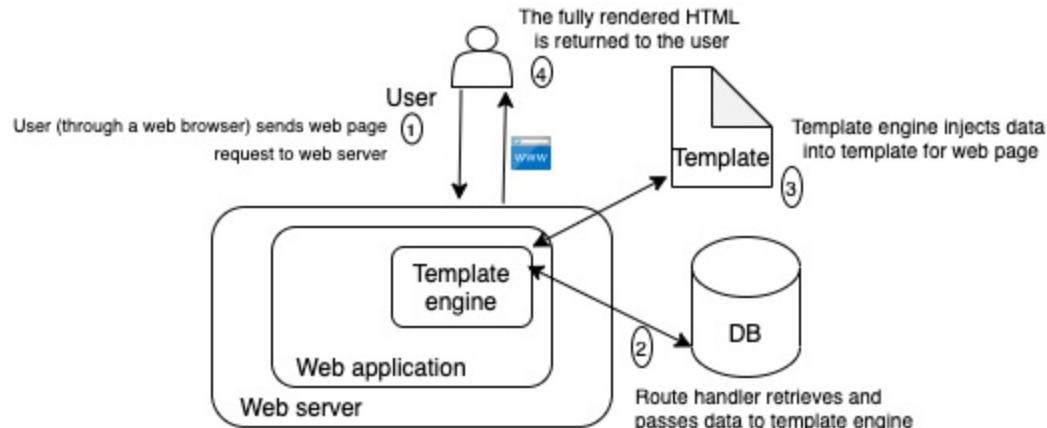


Figure 7.1 shows the various components that go into rendering a dynamic web page.

A *template engine* is one of the primary tools to convert a static web page into a dynamic web page. It expects a *template* file as input and generates an HTML file as output. In the process it embeds data (passed to it by the web application) into the *template* file to generate an HTML file. This process is dynamic in two ways. Firstly, the data is loaded on demand. Secondly, the data is tailored to the individual user requesting the data.

For developing *server-side web apps* in Rust, we will use the following tools/components:

1. *Actix web server* which will host a *web application* running at a specific port on the server, and route requests to the *handler* functions provided by the web application.
2. A *web application* written in Rust and deployed on the Actix web server, that will serve content in response to requests from a browser. This will contain the core *handler* logic that knows how to respond to various types of HTTP requests.
3. *Tera*, a template engine that's popular in the python world, and has been

ported to Rust.

4. Our own backend *Tutor web service* which we developed in the previous section, which will fetch the data from the database, and manage database interactions. The *web application* will talk to the *tutor web service* to retrieve data and perform transactions, rather than dealing with the database itself.
5. Built-in HTTP client from the Actix web framework, to talk to the *tutor web service*.

If the concept of *server-side rendering* (SSR) is still a bit clear to you, why don't we learn SSR with Rust by actually writing out some example code? If a picture is worth a thousand words, then even a few lines of code are worth several times that.

Note that this chapter is about learning how to build a web application by looking at smaller snippets of code, and understanding how the various pieces fit together to construct a web application. However, it is only in the next chapter that we will actually design and build the *tutor web application*. Here is a quick mind map of the examples you will be building in this chapter. These examples represent the most common tasks in any web application that allows users to view and maintain data from a browser-based user interface.

1. *Section 7.1* will show how to serve static web pages with *Actix web*.
2. *Section 7.2* will cover generation of dynamic web pages using *Tera*, a popular template engine in the web development world.
3. In *Section 7.3*, you'll learn to capture user input with an HTML form.
4. *Section 7.4* is about displaying lists of information using *Tera* HTML templates
5. You earlier learnt how to write automated tests for the *web service* (server-side), in *section 7.5* you'll learn to write client-side tests.
6. We'll conclude the chapter in *section 7.6* by connecting the *front-end web application* with the *backend web service* using an HTTP client.

With this background, let's get to the first section.

7.1 Serving a static web page with Actix

In the previous chapters, we used the Actix web server to host our *tutor web service*. In this first section of this chapter, we'll use Actix to serve a static web page. Consider this as the 'Hello World' program for web application development.

Let's first setup the project structure:

1. Make a copy of the *ezytutors* workspace repo from *Chapter 6*, to work within this chapter.
2. Create a new Rust cargo project with `cargo new tutor-web-app-ssr`
3. Rename *tutor-db* folder under *ezytutors* workspace to *tutor-web-service*. This way the two repos under the workspace can be referred to unambiguously as *web service* and *web app*.
4. In *Cargo.toml* of the workspace folder, edit the workspace section to look like this:

```
[workspace]
members = ["tutor-web-service", "tutor-web-app-ssr"]
```

We now have two projects in the workspace, one for the *tutor web service* (which we developed earlier) and another for the *tutor web app* that is rendered server-side (which we are yet to develop).

5. `cd tutor-web-app-ssr`

Switch to the *tutor-web-app-ssr* folder. That's where we'll write the code for this section. Henceforth, let's refer to this folder as the project root folder. To avoid confusion , set this as an environment variable in each of the terminal sessions you will be working with for this project, as shown:

```
export $PROJECT_ROOT=.
```

6. Update *Cargo.toml* to add the following dependencies.

```
[dependencies]
actix-web = "4.2.1"
actix-files="0.6.2"
```

actix-web is the core actix web framework and *actix-files* helps in serving static files from the web server.

7. Create a *static* folder under \$PROJECT_ROOT. Create a file *static-web-page.html* under \$PROJECT_ROOT/static with the following html code.

```
<!DOCTYPE html>
<html>
<head>
    <title>XYZ Bank Website</title>
</head>
<body>
    <h1>Welcome to XYZ bank home page!</h1>
    <p>This is an example of a static web page served from Actix We
</body>
</html>
```

This is a simple static web page. We'll see how to serve this page with the Actix server.

8. Create a *bin* folder under \$PROJECT_ROOT/src. Create a new source file *static.rs* under \$PROJECT_ROOT/src/bin and add the following code:

```
use actix_files as fs;                      #1
use actix_web::{error, web, App, Error, HttpResponse, HttpServer,         

#[actix_web::main]
async fn main() -> std::io::Result<()> {      #2
    let addr = env::var("SERVER_ADDR").unwrap_or_else(|_| "127.0.
    println!("Listening on: {}", open browser and visit have a try
    HttpServer::new(|| {
        App::new().service(fs::Files::new("/static", "./static")).
    })
    .bind(addr)?                                #4
    .run()                                       #5
    .await                                       #6
}
```

This program creates a new web application, registers a service with the web application to serve files from the file system (on disk), when a GET request is made to the web server on the route starting with */static*. The web application is then deployed on the web server, and the web server is started.

9. Run the web server with cargo run --bin static.
10. From a browser, visit the following url:

`http://localhost:8080/static/static-web-page.html`

You should see the web page appear in your browser.

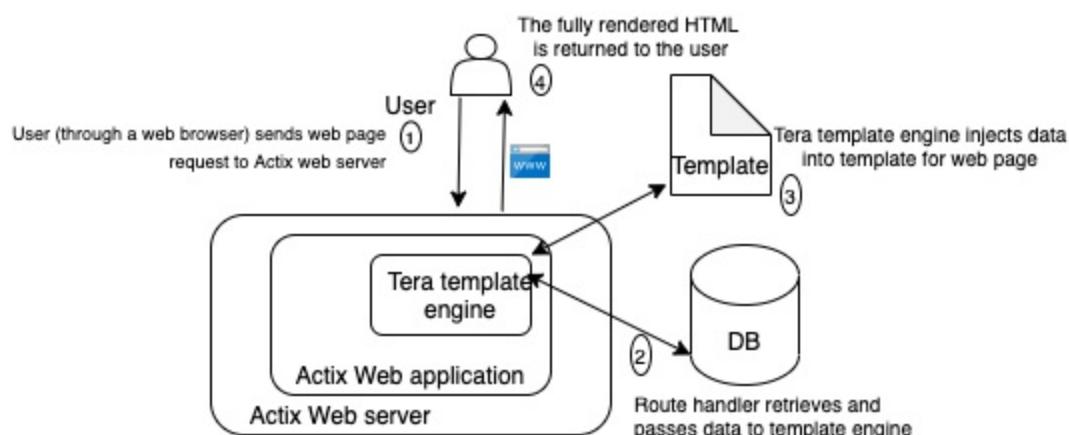
Let's now try to understand what we just did. We wrote a program to serve a static web page from an Actix web server. When we requested a particular static file, the `actix_files` service looked for it within the `/static` folder and returned it to the browser, which was then displayed to the user.

This is an example of a static page, because the content of this page does not change depending on the user who requests this page. In the next section, we'll see an example of how to build dynamic web pages with Actix.

7.2 Rendering a dynamic web page with Actix and Tera

What if we wanted to show custom content for each user? How would you write an HTML page which presents content dynamically? Note that displaying a dynamic web page does not mean everything in the page changes for every user, but that the web page has both static and dynamic parts to it.

Figure 7.2. Dynamic web pages with Actix and Tera



We've earlier seen a generic view of server-side rendering in *figure 7.1*. *Figure 7.2* shows how *server-side rendering* of dynamic web pages can be implemented using *Actix web* and *Tera* template engine. Note that in the

figure, a local database is shown as a source of data for the dynamic web page, but it is also possible to retrieve data from an external *web service*. In fact, this is the design approach that we will use in this book.

For this, we will define the HTML file in a specific template format. Details of the *Tera* template format can be viewed at: <https://tera.netlify.app/docs/>. Here is an example of a very simple template. Add this to `$PROJECT_ROOT/static/iter1/index.html`.

```
<!DOCTYPE html>
<html>

<head>
    <title>XYZ Bank Website</title>
</head>

<body>
    <h1>Welcome {{ name }}, to XYZ bank home page!</h1>
    <p>This is an example of a dynamic web page served with Actix a
</body>
</html>
```

Note the use of the tag `{{name}}`. This tag is substituted by Tera at run-time with the actual name of the user, when the web page is requested by the browser. Tera can retrieve this value from wherever you want it to - from a file, a database, or simply hard-coded values.

Let's modify the program we wrote earlier to cater to such dynamic web page requests using Tera.

In `$PROJECT_ROOT/Cargo.toml` add the following dependencies:

```
tera = "1.17.0"
serde = { version = "1.0.144", features = ["derive"] }
```

We're adding the *tera* crate for templating support and *serde* crate to enable custom data structures to be serialized/deserialized between the web browser and the web server.

In `$PROJECT_ROOT/src/bin`, copy the contents of the file `static.rs` we wrote earlier into a new file `iter1.rs`, and modify the following code to look like

this:

```
use tera::Tera;

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

    println!("Listening on: 127.0.0.1:8080, open browser and visi
HttpServer::new(|| {
    let tera = Tera::new(concat!(#1
        env!("CARGO_MANIFEST_DIR"),
        "/static/iter1/**/*"
    ))
    .unwrap();

    App::new()
        .data(tera) #2
        .service(fs::Files::new("/static", "./static").show_f
        .service(web::resource("/").route(web::get().to(index
    }))
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

Let's now write the *index* handler:

```
async fn index tmpl: web::Data<tera::Tera> ) -> Result<HttpResponse>
    let mut ctx = tera::Context::new(); #1
    ctx.insert("name", "Bob"); #2
    let s = tmpl
        .render("index.html", &ctx) #3
        .map_err(|_| error::ErrorInternalServerError("Template er
    Ok(HttpResponse::Ok().content_type("text/html").body(s))
}
```

Run the server with `cargo run --bin iter1`. Then from a web browser access the following URL:

`http://localhost:8080/`

You should see the following message displayed on the web page:

Welcome Bob, to XYZ bank home page!

This is a trivial example, but serves to illustrate the concept of how dynamic web pages can be constructed using Actix. The Tera website listed earlier has a lot of features that can be used as part of the template including control statements such as *if* and *for* loops, which you can explore at leisure.

We've so far seen how to render both static web(HTML) pages and dynamic HTML pages. But the examples so far dealt with displaying some information to a user. Does Actix also support writing HTML pages that accept user input? We'll find out in the next section.

7.3 Adding user input with forms

In this section we'll create a web page that accepts user inputs through a form. Here is a form that's as simple as it can get. Create a folder \$PROJECT_ROOT/static/iter2 and place the following html in a new file *form.html* under this folder. This html code contains a form that accepts a tutor name, and then submits a POST request containing the tutor name, to the Actix web server.

```
<!doctype html>
<html>

<head>
    <meta charset=utf-8>
    <title>Forms with Actix & Rust</title>
</head>

<body>
    <h3>Enter name of tutor</h3>
    <form action=/tutors method=POST>
        <label>
            Tutor name:
            <input name="name">
        </label>
        <button type=submit>Submit form</button>
    </form>

    <hr>
</html>
```

Note the *<input>* element of HTML that is used to accept user input for a

tutor name. The `<button>` tag is used to submit the form to the web server. This form is encapsulated in an HTTP POST request sent to the web server on the route `/tutors`, which is specified in the `<form action="">` attribute.

Let's create a second html file under the `$PROJECT_ROOT/static/iter2` folder called `user.html`, which will display the name submitted by the user in the previous form.

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Actix web</title>
</head>

<body>
    <h1>Hi, {{ name }}!</h1>
    <p>
        {{ text }}
    </p>
</body>

</html>
```

This HTML file has a template variable `{{name}}`. When this page is shown to the user, the value of the template variable `{{name}}` is replaced with the actual *tutor name* that was entered by the user in the previous form.

Let's now add this route, and also a handler to deal with this POST request.

In `$PROJECT_ROOT/src/bin`, create a new file `iter2.rs`, and add the following code to `iter2.rs`.

```
... // imports removed for concision; see full source code from G

// store tera template in application state
async fn index() #1
    tmpl: web::Data<tera::Tera>
) -> Result<HttpResponse, Error> {
    let s = tmpl
        .render("form.html", &tera::Context::new()) #2
        .map_err(|_| error::ErrorInternalServerError("Template er
```

```

        Ok(HttpResponse::Ok().content_type("text/html").body(s))
    }

#[derive(Serialize, Deserialize)] #3
pub struct Tutor {
    name: String,
}

async fn handle_post_tutor( #4
    tmpl: web::Data<tera::Tera>,
    params: web::Form<Tutor>,
) -> Result<HttpResponse, Error> {
    let mut ctx = tera::Context::new();
    ctx.insert("name", &params.name); #5
    ctx.insert("text", "Welcome!");
    let s = tmpl
        .render("user.html", &ctx)
        .map_err(|_| error::ErrorInternalServerError("Template er

        Ok(HttpResponse::Ok().content_type("text/html").body(s))
    }

#[actix_web::main] #6
async fn main() -> std::io::Result<()> {
    println!("Listening on: 127.0.0.1:8080");
    HttpServer::new(|| {
        let tera = Tera::new(concat!(
            env!("CARGO_MANIFEST_DIR"),
            "/static/iter2/**/*"
        ))
        .unwrap();

        App::new() #7
            .data(tera)
            .configure(app_config)
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

fn app_config(config: &mut web::ServiceConfig) { #8
    config.service(
        web::scope("")
            .service(web::resource("/").route(web::get().to(index)))
            .service(web::resource("/tutors").route(web::post().to(hand
));
}

```

```
}
```

To recap, in the code shown, when a user visits the "/" route, the *form.html* is displayed which contains a form. When the user enters the name in the form and presses the submit button, a *POST* request is generated on route */tutors*, which invokes another handler function *handle_post_tutor*. In this handler, the name entered by the user is accessible through the *web::Form* extractor. The handler injects this name into a new *Tera context* object. The *Tera* render function is then invoked with the context object, to show *user.html* page to the user.

Run the web server with:

```
cargo run --bin iter2
```

From a browser access the URL:

```
http://localhost:8080/
```

You should first see the form displayed. Enter a name and click the *Submit form* button. You should see the second html displayed containing the name you entered.

This concludes this section on demonstrating how you can accept user inputs and process it. In the next section, we'll cover another common feature of the template engine - ability to display lists.

7.4 Displaying a list with templates

In this section, we'll learn how to display a list of data elements dynamically on a web page. In the *tutor web app*, one of the things a user would want is to see a list of tutors or courses. This list is dynamic because the user may either want to see a list of all tutors in the system, or a subset of tutors based on some criteria. Likewise, the user may want to see a listing of all courses available on the site or the courses for a particular tutor. How would we use Actix and Tera to show such information? Let's find out.

Create a folder *iter3* under *\$PROJECT_ROOT/static*. Create a new file

list.html here and add the following html.

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Actix web</title>
</head>

<body>
    <h1>Tutors list</h1>
    <ol>
        {% for tutor in tutors %}                                #1
        <li>                                              #2
            <h5>{{tutor.name}}</h5>                            #3
        </li>
        {% endfor %}                                         #4
    </ol>
</body>

</html>
```

To summarize, we have written an HTML file that contains a template control statement (using a *for* loop) which loops through each tutor in a list and displays the tutor name on the web page.

Next, let's write the *handler* function to implement this logic, and the *main* function for the web server.

Create a new file *iter3.rs* under *\$PROJECT_ROOT/src/bin* and add the following code:

```
use actix_files as fs;
use actix_web::{error, web, App, Error, HttpResponse, HttpServer,
use serde::{Deserialize, Serialize};
use tera::Tera;

#[derive(Serialize, Deserialize)]
pub struct Tutor {                                         #1
    name: String,
}

async fn handle_get_tutors(tmp: web::Data<tera::Tera>) -> Result
    let tutors: Vec<Tutor> = vec![                      #3
```

```

        Tutor {
            name: String::from("Tutor 1"),
        },
        ...
    ];
    let mut ctx = tera::Context::new();                      #
    ctx.insert("tutors", &tutors);                          #
    let rendered_html = tmpl                            #
        .render("list.html", &ctx)                         #
        .map_err(|_| error::ErrorInternalServerError("Template er
Ok(HttpResponse::Ok().content_type("text/html").body(rendered
}

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    println!("Listening on: 127.0.0.1:8080");
    HttpServer::new(|| {
        let tera = Tera::new(concat!(
            env!("CARGO_MANIFEST_DIR"),
            "/static/iter3/**/*"
        ))
        .unwrap();

        App::new()
            .data(tera)
            .service(fs::Files::new("/static", "./static").show_f
            .service(web::resource("/tutors").route(web::get().to

    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}

```

Run the web server with:

```
cargo run --bin iter3
```

From a web browser, access the following URL:

```
http://localhost:8080/tutors
```

You should see the list of tutors displayed.

After the initial euphoria of seeing the tutor list displayed has waned, you will start to notice that the web page isn't particularly impressive or aesthetic. You would most certainly now want to add some css to the web page. Sure, it can be done easily. Here is an example css just for illustration purposes. Place this code in `styles.css` under `/static` folder, which we have already declared in the main function to be the source of static assets.

```
/* css */
ul {
    list-style: none;
    padding: 0;
}
li {
    padding: 5px 7px;
    background-color: #FFEBBC;
    border: 2px solid #DEB887;
}
```

In `list.html` under `$PROJECT_ROOT/iter3__`, add the css file to the head block of html as follows:

```
<head>
    <meta charset="utf-8" />
    <link rel="stylesheet" type="text/css" href="/static/styles.css"/>
    <title>Actix web</title>
</head>
```

Run the web server again and visit the `/tutors` route from a web browser. You should now see the css styles reflect on the web page. This may still not be the prettiest of pages, but you now understand how you can add your own styling to the web page.

But if you're like me, and don't want to write your own custom css, you can import one of your preferred css frameworks like this. Change the HEAD section of `list.html` file to import `tailwind.css`, a popular modern css library. You can import `Bootstrap`, `Foundation`, `Bulma`, or any other css framework of choice.

```
<!DOCTYPE html>
<html>

<head>
```

```

<meta charset="utf-8" />
<title>Actix web</title>
<link href="https://unpkg.com/tailwindcss@^1.0/dist/tailwind.css" rel="stylesheet">
</head>

<body>
    <h1 class="text-2xl font-bold mt-8 mb-5">Tutors list</h1>
    <ul class="list-disc list-inside my-5 pl-2">
        {% for tutor in tutors %}
            <ol class="list-decimal list-inside my-5 pl-2">
                <h5 class="text-1xl font-bold mb-4 mt-0">{{tutor.name}}</h5>
            </ol>
        {% endfor %}
    </ul>
</body>

</html>

```

Compile and run the server again, and this time you should see something hopefully a little more appealing to your eye.

We will not spend much time on CSS styles in this book, but CSS being an integral part of web pages, it is important for you to know how to use it with Actix and templates.

We've so far seen different ways to show dynamic content in web pages using Actix and Tera. Let's now shift gears and focus on one more important aspect of developing front-end web apps: *automated unit and integration tests*. Just as we were able to write test cases for the *backend tutor web service*, is it also possible to write test cases for the front-end web app in Rust with Actix and tera? Let's find out in the next section.

7.5 Writing and running client-side tests

For this section, we'll not be writing any new application code, but instead reuse one of the handler functions that we've previously written, and learn how to write unit test cases for the handler.

Let's use the code we wrote in *iter2.rs*. Specifically, here is the handler function that we'll focus on:

```

async fn handle_post_tutor(
    tmpl: web::Data<tera::Tera>,
    params: web::Form<Tutor>,
) -> Result<HttpResponse, Error> {
    let mut ctx = tera::Context::new();
    ctx.insert("name", &params.name);
    ctx.insert("text", "Welcome!");
    let s = tmpl
        .render("user.html", &ctx)
        .map_err(|_| error::ErrorInternalServerError("Template er
                                          ok(HttpResponse::Ok().content_type("text/html").body(s)))
}

```

This handler can be invoked from the command-line using a curl POST request as shown.

```
curl -X POST localhost:8080/tutors -d "name=Terry"
```

Let's write a unit test case for this handler function.

In *\$PROJECT_ROOT/Cargo.toml*, add the following section:

```
[dev-dependencies]
actix-rt = "2.2.0"
```

actix-rt is the Actix async runtime, which is needed to execute the asynchronous test functions.

In *\$PROJECT_ROOT/src/bin/iter2.rs*, add the following test code towards the end of the file (as a convention, the Rust unit test cases are located towards the end of the source file).

```

#[cfg(test)]                                     #1
mod tests {                                     #2
    use super::*;

    use actix_web::http::{header::CONTENT_TYPE, HeaderValue, Status};
    use actix_web::web::Form;
}

#[actix_rt::test]                                #3
async fn handle_post_1_unit_test() {              #4
    let params = Form(Tutor {
        name: "Terry".to_string(),

```

```

    });
    let tera = Tera::new(concat!(
        env!("CARGO_MANIFEST_DIR"),
        "/static/iter2/**/*"
    ))
    .unwrap();
    let webdata_tera = web::Data::new(tera);           #6
    let resp = handle_post_tutor(webdata_tera, params).await.

    assert_eq!(resp.status(), StatusCode::OK);
    assert_eq!(
        resp.headers().get(CONTENT_TYPE).unwrap(),
        HeaderValue::from_static("text/html")
    );
}
}
}

```

Run the tests from `$PROJECT_ROOT` with:

```
cargo test --bin iter2
```

You should see that the test passes.

We've just written a unit test case by invoking the handler function directly. We were able to do it because we know the handler function signature. This is ok for a unit test case, but how would we simulate a web client posting an HTTP request with the form data?

That's the domain of integration testing. Let's write an integration test case to simulate a user form submission.

Add the following to `tests` module in `$PROJECT_ROOT/src/bin/iter2.rs`.

```

use actix_web::dev::{HttpResponseBuilder, Service, ServiceRes
use actix_web::test::{self, TestRequest};

// Integration test case
#[actix_rt::test]
async fn handle_post_1_integration_test() {
    let tera = Tera::new(concat!(
        env!("CARGO_MANIFEST_DIR"),
        "/static/iter2/**/*"
    ))
    .unwrap();

```

```

let mut app = test::init_service(App::new().data(tera).co

let req = test::TestRequest::post()                      #2
    .uri("/tutors")
    .set_form(&Tutor {
        name: "Terry".to_string(),
    })
    .to_request();                                     #3
let resp: ServiceResponse = app.call(req).await.unwrap();
assert_eq!(resp.status(), StatusCode::OK);               #
assert_eq!(
    resp.headers().get(CONTENT_TYPE).unwrap(),          #
    HeaderValue::from_static("text/html")
);
}

```

You'll notice that Actix provides rich support for testing in the form of built-in services, modules and functions, which we can use to write unit or integration tests.

Run the tests from `$PROJECT_ROOT` with:

```
cargo test --bin iter2
```

You should see both unit and integration tests pass.

With this, we conclude the section on learning to write unit and integration test cases for front-end web apps built with Actix and tera. We'll be using what we have learnt here, to write the actual test cases while developing the tutor web application.

7.6 Connecting to the backend web service

In a previous section, we displayed a list of tutors on a web page using mock data. In this section, we'll fetch data from the backend *tutor web service* to display on the web page instead of mock data. Note that technically, we can directly talk to a database from the Actix web application, but that's not what we want to do. The main reason is that we do not want to duplicate the database access logic that is already present in the web service. Another reason is that we do not want to expose the database access credentials in both the web service and web application, which could increase the surface

area of any security/hacking attacks.

We know that the backend *tutor web service* exposes various REST APIs. To talk to the web service from the web application, we need an HTTP client that can be embedded within the web application. While there are other external crates available for this, let's use the built-in HTTP client in the Actix-web framework. We also need a way to parse and interpret the json data that is returned from the web service. For this, we'll use the *serde_json* crate.

Add the following to *\$PROJECT_ROOT/Cargo.toml*:

```
serde_json = "1.0.64"
```

Let's now write the code to connect to make a GET request to the *tutor web service* and retrieve the list of tutors.

Create a new file *iter4.rs* under *\$PROJECT_ROOT/src/bin* and copy the contents of *iter3.rs* to it, to get a headstart.

Using *serde_json* crate, we can deserialize the incoming json payload in HTTP response into a strongly typed data structure. In our case, we want to convert the json sent by the *tutor web service* into a *Vec<Tutor>* type. We would also like to define the structure of the *Tutor* struct to match the incoming json data. Remove the old definition of *Tutor* struct in the file *\$PROJECT_ROOT/src/bin/iter4.rs* and replace it with the following:

```
#[derive(Serialize, Deserialize, Debug)]
pub struct Tutor {
    pub tutor_id: i32,
    pub tutor_name: String,
    pub tutor_pic_url: String,
    pub tutor_profile: String,
}
```

Within the same source file, in the *handle_get_tutors* handler function, let's connect to the *tutor web service* to retrieve the tutor list. In that case, we can remove the hardcoded values. Import the *actix_web client module* and modify the code for the *handle_get_tutors* handler function as shown:

```

use actix_web::client::Client;

async fn handle_get_tutors(tmpl: web::Data<tera::Tera>) -> Result
    let client = Client::default(); #1

    // Create request builder and send request

    let response = client
        .get("http://localhost:3000/tutors/") #2
        .send() #3
        .await #4
        .unwrap() #5
        .body() #6
        .await #4
        .unwrap(); #5

    let str_list = std::str::from_utf8(&response.as_ref().unwrap())
    let tutor_list: Vec<Tutor> = serde_json::from_str(str_list).u
    let mut ctx = tera::Context::new();

    ctx.insert("tutors", &tutor_list);
    let rendered_html = tmpl
        .render("list.html", &ctx)
        .map_err(|_| error::ErrorInternalServerError("Template er

            Ok(HttpResponse::Ok().content_type("text/html").body(rendered
}

```

The rest of the code related to rendering Tera templates is similar to what we've seen before.

Next, create a new folder `$PROJECT_ROOT/static/iter4`. Under this folder place a copy of the `list.html` file from `$PROJECT_ROOT/static/iter3`. Alter the `list.html` file to change the template variable `{{tutor.name}}` to `{{tutor.tutor_name}}`, because that's the structure of the data sent back from the `tutor` web service.

Here is the updated `list.html` listing, under `iter4` folder.

```

<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8" />
    <title>Actix web</title>

```

```

        <link href="https://unpkg.com/tailwindcss@^1.0/dist/tailwind.
</head>

<body>
    <h1 class="text-2xl font-bold mt-8 mb-5">Tutors list</h1>
    <ul class="list-disc list-inside my-5 pl-2">
        {% for tutor in tutors %}
        <ol class="list-decimal list-inside my-5 pl-2">
            <h5 class="text-1xl font-bold mb-4 mt-0">{{tutor.tutor_</ol>
        {% endfor %}
    </ul>
</body>

</html>

```

Also alter the *main()* function in *iter4.rs* to look for Tera templates in *\$PROJECT_ROOT/static/iter4* folder. Here is the updated *main()* function.

```

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    println!("Listening on: 127.0.0.1:8080!");
    HttpServer::new(|| {
        let tera = Tera::new(concat!(env!("CARGO_MANIFEST_DIR"), "/sta
            App::new()
                .data(tera)
                .service(fs::Files::new("/static", "./static").show_files_1
                    .service(web::resource("/tutors").route(web::get().to(handl
            })
            .bind("127.0.0.1:8080")?
            .run()
            .await
    }
}

```

What we have done so far is to fetch the tutor list from the tutor web service (instead of the hard-coded values used in iteration 3), and use it to display the tutor list in *list.html* file, which is rendered when an HTTP request arrives from a client at route */tutors*.

To test this, first go to folder *tutor_web_service* under the *ezytutors* workspace, and run the server in a separate terminal. This server should now be listening on localhost:3000. Test the server with the following command:

```
cargo run --bin iter6
```

iter6 was the last iteration we built for the *tutor web service*.

Then from another terminal, run the *tutor_ssr_app* web server from *\$PROJECT_ROOT* with the following command:

```
cargo run --bin iter4
```

We now have the *tutor web service* running on port 3000 and *tutor web app* running on port 8080, both on localhost. Here's what should happen: when the user visits the */tutors* route on port 8080, the request would go to the web handler of the web app, which then would call out to the *tutor web service* to retrieve the tutor list. The *tutor web app* handler would then inject this data into tera template and display the web page back to the user.

To test this from a browser, visit the URL:

```
localhost:8080/tutors
```

You should see the list of tutor names populated in the web page, which was retrieved from our *tutor web service*. If you have reached this far, congratulations! If you encounter any errors, just retrace the code back to the last point when you had it working, and reapply the changes in sequence again by following the appropriate instructions in this chapter.

With this we have learnt the critical aspects of developing a client-side application with Actix. In the next chapter we will use the knowledge and skills gained in this chapter, to write the code for the *tutor web application*.

7.7 Summary

- In this chapter, we covered the basics of working with Actix to develop a server-side web application.
- We first learnt how to serve static web pages using Actix.
- In the second section, we built a simple dynamic web page using Actix and Tera templates. We learnt how to inject Tera into the web application and make it available to all the handlers. We also learnt how

to create a Tera context object, insert data into it, and render the Tera html template by passing on the values for the template variables defined in Tera template.

- In the third section, we learnt how to accept user inputs through a form, and trigger an HTTP request on a specific route of the web application, on submission of the form by the user.
- In the fourth section, we learnt how to render a list of tutor names in a web page using Tera templates.
- We then learnt how to write unit and integration test cases for the web application handlers.
- In the final section, we connected to the backend tutor web service api, and retrieved the list of tutors. The tutor list was displayed to the user.
- Rust can be used to not just build backend *web services*, but also front-end *web applications*.
- Server-side rendering (SSR) is a web architectural pattern that involves creating a fully-rendered web page on the server and simply sending it to the browser for display. SSR typically involves serving a mix of static and dynamic content on a web page.
- *Actix Web* and *Tera* template engine are powerful tools to implement *server-side rendering* in Rust-based web applications.
- *Tera* template engine is instantiated and injected into the web application in the `main()` function. The `tera` instance is made available to all the handler functions by the *Actix* web framework. The route handler functions, in turn, can use `tera` templates to construct dynamic web pages that are sent back to the browser client as part of the HTTP response body.
- HTML forms are used to capture user inputs, and post those inputs to a route on the *Actix* web application. The corresponding route handler then processes that HTTP request and sends back an HTTP response containing the dynamic web page.
- The control flow features of *Tera* templates can be used to display lists of information on a web page. The contents of the list can be retrieved either from a local database or an external web service, and injected into the web page template.
- *Actix Web Client* can be used as an HTTP client to communicate between the *Actix web application front-end* and *Actix web service backend*.

In the next part of the book, we'll jump straight into writing the tutor web application that can act as a client front-end to the tutor web service.

See you in the next chapter.

8 Working with templates for tutor registration

This chapter covers

- Designing tutor registration feature
- Setting up the project structure
- Displaying the registration form
- Handling registration submission

In the previous chapter, we covered the basics of working with Actix to develop a server-side web application. In this chapter we'll learn more details of how to work with templates, by creating a tutor registration form using Actix and Terra.

Templates and forms are an important feature of web applications. They are used quite commonly for registration, sign in, capturing user profile, payment information or KYC (know-your-customer) details for regulatory purposes and performing CRUD (create-read-update-delete) operations on data. While capturing user inputs, it is also necessary to validate them and provide feedback to the user in case of errors. In cases where the forms involve data updates, existing information has to be presented to the user in the form, allowing the user to change it. There are also elements of styling to be added for aesthetic appeal. On submission of forms, the form data needs to be serialized into an HTTP request which should then invoke the right handler functions for processing and storing the form data. Finally, the user needs to be given feedback on the success of the form submission, and then optionally taken to the next screen. We'll learn how to do all these in this chapter using Actix Web, Tera template engine and a few other components.

Let's first start with the design of what we will be building here.

8.1 Designing tutor registration feature

In this chapter we'll write an html template and associated code to allow tutors to register.

Figure 1 shows the tutor registration form.

Figure 8.1. Tutor registration form

Tutor registration

Enter username

Enter password

Confirm password

Enter tutor name

Enter tutor image url

Brief tutor profile

Register

Sign in

For registration, we'll accept six fields: *username*, *password*, *password confirmation*, *tutor name*, *tutor image url*, and *brief tutor profile*. The first three will be used for user management functions, and the others will be used to send the request to the *tutor web service* to create a new tutor in the database.

Let's first setup the project code structure and basic scaffolding.

8.2 Setting up the project structure

First copy/clone the code from Chapter 7. We'll build on this code structure. Navigate to folder *tutor-web-app-ssr* under *ezytutors*. This represents the project root.

Let's also set the *PROJECT_ROOT* environment variable to */path-to-folder/ezytutors/tutor-web-app-ssr*. Henceforth, we'll refer to this folder as *\$PROJECT_ROOT*.

Let's organize the code under the *PROJECT_ROOT* as follows:

1. Create a folder *iter5* under *\$PROJECT_ROOT/src*. This will contain the data model, routes handler functions, definitions for *custom error type* and *application state*, and database sql scripts.
2. Create a folder *iter5* under *\$PROJECT_ROOT/static*. This folder will contain the html/terra templates.
3. Create a file *iter5-ssr.rs* under *\$PROJECT_ROOT/bin*. This is the main function that will configure and startup the Actix web server (to serve the web application that we are building).
4. Under *\$PROJECT_ROOT/src/iter5*, create the following files:
 - *routes.rs*: Stores the routes for the web application on which HTTP requests can be received.
 - *model.rs*: Contains the data model definitions.
 - *handler.rs*: Contains the handler functions associated with the various routes, to process the incoming HTTP requests.
 - *state.rs*: To store the data structure representing the application

- state, which will be injected into the handlers (aka *dependency injection*)
- *errors.rs*: Contains the custom error type and associated functions to construct suitable error messages for users
 - *dbaccess.rs*: Contains the functions that access the database for reading and writing tutor data.
 - *dbscripts/user.sql*: Create a folder *dbscripts* under *\$PROJECT_ROOT/src/iter5*, and create a file *user.sql* under it. This will contain the sql scripts to create a database table.
 - *mod.rs*: To configure the *\$PROJECT_ROOT/src/iter5* directory as a Rust module that can be imported into other files.

We’re now ready to start coding.

Let’s begin with the routes definition in *\$PROJECT_ROOT/src/iter5/routes.rs*.

```
use crate::handler::{handle_register, show_register_form}; #1
use actix_files as fs;                                     #2
use actix_web::web;

pub fn app_config(config: &mut web::ServiceConfig) {          #3
    config.service(
        web::scope("")
            .service(fs::Files::new("/static", "./static").show_files_1)
            .service(web::resource("/").route(web::get().to(show_register)))
            .service(web::resource("/register").route(web::post().to(handle_register)));
    }
}
```

With this, we can move on to the model definition in *\$PROJECT_ROOT/src/iter5/model.rs*.

Add the following data structures to *model.rs*:

Listing 8.1. Data Model

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

#[derive(Serialize, Deserialize, Debug)]
pub struct TutorRegisterForm {                                #1
```

```

    pub username: String,
    pub password: String,
    pub confirmation: String,
    pub name: String,
    pub imageurl: String,
    pub profile: String,
}

#[derive(Serialize, Deserialize, Debug)]
pub struct TutorResponse {                                     #2
    pub tutor_id: i32,
    pub tutor_name: String,
    pub tutor_pic_url: String,
    pub tutor_profile: String,
}

#[derive(Serialize, Deserialize, Debug, sqlx::FromRow)]
pub struct User {                                         #3
    pub username: String,
    pub tutor_id: Option<i32>,
    pub user_password: String,
}

```

Let's next define the application state in `$PROJECT_ROOT/src/iter5/state.rs`.

```

use sqlx::postgres::PgPool;

pub struct AppState {
    pub db: PgPool,
}

```

The AppState will hold the Postgres connection pool object, which will be used by the database access functions. The AppState will be injected into each handler function by Actix-web, we'll see later how to configure this while creating the Actix application instance.

Let's also create an `error.rs` file under `$PROJECT_ROOT/src/iter5` to define a custom error type. This is mostly similar to the error definition we earlier created for the tutor web service, but with some minor changes.

Listing 8.2. Custom error type

```
use ... #1
```

```

#[derive(Debug, Serialize)]
pub enum EzyTutorError { #2
    DBError(String),
    ActixError(String),
    NotFound(String),
    TeraError(String),
}
#[derive(Debug, Serialize)]
pub struct MyErrorResponse { #3
    error_message: String,
}
impl std::error::Error for EzyTutorError {} #4

impl EzyTutorError { #5
    fn error_response(&self) -> String {
        match self {
            EzyTutorError::DBError(msg) => {
                println!("Database error occurred: {:?}", msg);
                "Database error".into()
            }
            EzyTutorError::ActixError(msg) => { ... } #1
            EzyTutorError::TeraError(msg) => { ... } #1
            EzyTutorError::NotFound(msg) => { ... } #1
        }
    }
}

impl error::ResponseError for EzyTutorError { #6
    fn status_code(&self) -> StatusCode {
        match self {
            EzyTutorError::DBError(_msg)
            | EzyTutorError::ActixError(_msg)
            | EzyTutorError::TeraError(_msg) => StatusCode::INTERNAL_ERROR
            EzyTutorError::NotFound(_msg) => StatusCode::NOT_FOUND
        }
    }
    fn error_response(&self) -> HttpResponse {
        HttpResponse::build(self.status_code()).json(MyErrorResponse {
            error_message: self.error_response(),
        })
    }
}

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

```

```

        }
    }

impl From<actix_web::error::Error> for EzyTutorError {
    fn from(err: actix_web::error::Error) -> Self {
        EzyTutorError::ActixError(err.to_string())
    }
}

impl From<SQLxError> for EzyTutorError { ... }

```

We've so far defined the routes, data model, application state and error type. Let's next write the scaffolding for the various handler functions. These won't do much, but will establish the code structure which we can build on in future sections.

In `$PROJECT_ROOT/src/iter5/handler.rs`, add the following:

```

use actix_web::{Error, HttpResponse, Result};

pub async fn show_register_form() -> Result<HttpResponse, Error>
{
    let msg = "Hello, you are in the registration page";
    Ok(HttpResponse::Ok().content_type("text/html").body(msg))
}

pub async fn handle_register() -> Result<HttpResponse, Error> {
    Ok(HttpResponse::Ok().body(""))
}

```

As you can see, the handler functions don't really do much, but it is sufficient for us to establish the initial code structure that we can build on.

Lastly, let's write the `main()` function that will configure the web application with the associated routes configuration, and launch the web server.

Add the following code to `$PROJECT_ROOT/bin/iter5-ssr.rs`.

Listing 8.3. `main()` function

```

#[path = "../iter5/mod.rs"]                                #1
mod iter5;                                              #1
use iter5::{dbaccess, errors, handler, model, routes, state::AppS
use routes::app_config;                                  #1

```

```

use actix_web::{web, App, HttpServer};                      #2
use dotenv::dotenv;                                         #3
use std::env;                                              #3
use sqlx::postgres::PgPool;                                #4

use tera::Tera;

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    dotenv().ok();                                         #5
    //Start HTTP server
    let host_port = env::var("HOST_PORT").expect("HOST:PORT addre
    println!("Listening on: {}", &host_port);
    let database_url = env::var("DATABASE_URL").expect("DATABASE_
    let db_pool = PgPool::connect(&database_url).await.unwrap();
    // Construct App State
    let shared_data = web::Data::new(AppState { db: db_pool });

    HttpServer::new(move || {                                     #8
        let tera = Tera::new(concat!(env!("CARGO_MANIFEST_DIR"), "/sta

            App::new()
                .data(tera)
                .app_data(shared_data.clone())
                .configure(app_config)
        })
        .bind(&host_port)?                                      #9
        .run()                                                 #9
        .await                                                 #9
    })
}

```

We have to do a couple of more things. First, add the dotenv package to *Cargo.toml* file in *\$PROJECT_ROOT*. Make sure the *Cargo.toml* file looks similar to this:

```

[dependencies]
actix-web = "4.2.1"
actix-files="0.6.2"
tera = "1.17.0"
serde = { version = "1.0.144", features = ["derive"] }
serde_json = "1.0.85"
awc = "3.0.1"
sqlx = {version = "0.6.2", default_features = false, features = [
rust-argon2 = "1.0.0"
dotenv = "0.15.0"

```

```
[dev-dependencies]
actix-rt = "2.7.0"
```

Configure the host, port and database details in the `.env` file in `$PROJECT_ROOT` as shown:

```
HOST_PORT=127.0.0.1:8080
DATABASE_URL=postgres://ssruser:mypassword@127.0.0.1:5432/eytutor_web_ss
```

The `DATABASE_URL` specifies the username(`ssruser`) and password(`mypassword`) for database access. It also specifies the port number at which the `postgres` database processes are running, and the name of the database (`eytutor_web_ss`) to connect to. We'll cover more details of this in a later section.

Lastly, add the following entries to `mod.rs` under `$PROJECT_ROOT/src/iter5`. This will export the functions and data structures we have defined and allow them to be imported and used elsewhere in the application.

```
pub mod dbaccess;
pub mod errors;
pub mod handler;
pub mod model;
pub mod routes;
pub mod state;
```

We're ready to test. Run the following from `$PROJECT_ROOT`

```
cargo run --bin iter5-ssr
```

You should see the Actix web server startup and listen on the specified host:port combination in the `.env` file.

From a browser, try the following URL route (adjust the port number to your own in `.env` file):

```
localhost:8080/
```

You should see the following message displayed on your browser screen:

Hello, you are in the registration page

We have now established the basic project structure and are ready to implement the logic to display registration form to the user.

8.3 Displaying the registration form

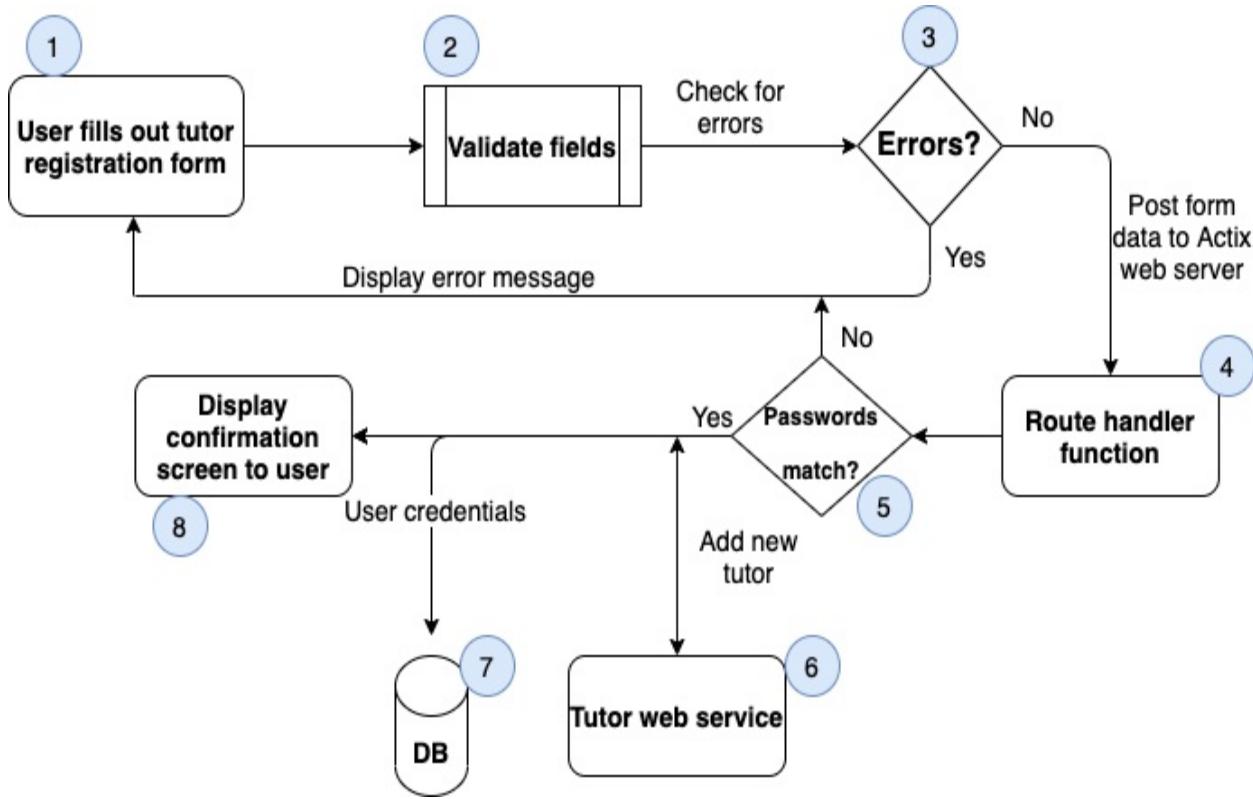
In earlier chapters, we have built the APIs on the *tutor web service* for adding, updating and deleting tutor information. We tested these APIs using command-line tools. What we're going to do in this chapter is to add the following two additional features:

1. Provide a web user interface where tutors can register
2. Store user credentials in a local database (for user management)

Note that on #2, user management can be done in different ways. It can be built directly into the backend *web service* or it can be handled in the front-end *web application*. In this chapter, we'll adopt the latter approach, mainly to demonstrate how to implement separation of responsibilities between the *backend web service* and *front-end web application* as a design choice. In this model, the *backend web service* takes care of the core *business and data access logic* to store and apply rules on tutor and course data, while the *front-end web application* handles the user authentication and session management functions. In such a design, we would have the *tutor web service* running in a trusted zone behind the firewall, receiving HTTP requests only from the trusted *front-end web application*.

Let's now take a look at the tutor registration workflow shown in figure 2.

Figure 8.2. Tutor registration flow



1. User visits the *landing page URL*. The web browser will make a *GET* request on index route '/', which is routed by Actix web server to the handler function `show_register_form()`. This function will send the registration form back to the web browser as an HTTP response. The *Tutor registration form* is now displayed to the user.
2. The user starts to fill out the registration form. There may be invalid inputs from users which need to be corrected (e.g. password does not meet minimum length criteria). How do we communicate this to the user?
3. For this, the HTML specifications allow us to do a few types of basic validation checks within the browser itself, rather than make a round-trip to the server every time. We'll make use of this to enforce mandatory field checks and field length checks, so that in case of errors in user input, feedback is provided to the user right within the web browser.
4. User completes and submits the *registration form*. A *POST* request is sent to the Actix web server on the `/register` route. The Actix web framework routes the request to the `handle_register()` web handler.
5. The `handle_register()` function checks to see if the *password* and *password confirmation* fields match. If they don't, the *registration form*

is displayed back to the user with an appropriate error message. This is a case of validating user input on the server rather than from within the browser which we performed in an earlier step. (*Note that it is possible to perform this validation using custom jQuery or javascript in the browser, but we're avoiding that approach in this book, if only to demonstrate that it is possible to write complete web applications in Rust without javascript. But you can use javascript, should you choose*)

6. If passwords match, the `handle_register()` function makes a *POST* request on the backend *tutor web service*, to create a new tutor entry in the database.
7. The *username* and *password* provided by the user in the registration form are stored in a local database on the *tutor web application* (note: not in *tutor web service*), for the purpose of authenticating the user in future.
8. A confirmation page is returned by the `handle_register()` function as HTTP response, to the web browser.

Now that we understand what we're going to build, let's start with the static assets and templates for tutor registration.

In `#PROJECT_ROOT/static/iter5/` create a file `register.html`, add the following contents.

Listing 8.4. Registration template

```
<!doctype html>
<html>

<head>
    <meta charset=utf-8>
    <title>Tutor registration</title>
    <link rel="stylesheet" href="/static/tutor-styles.css">      #
</head>

<body>                      #2
    <div class="header">
        <h1>Welcome to EzyTutor</h1>
        <p>Start your own online tutor business in a few minutes<
    </div>
    <div class="center">
        <h2>
```

```

        Tutor registration
</h2>
<form action=/register method=POST>          #3
    <label for="userid">Enter username</label><br>
    <input type="text" name="username" value="{{current_u
        maxlength="12" required><br>          #4
    <label for="password">Enter password</label><br>
    <input type="password" name="password" value="{{curre
        minlength="8" maxlength="12" required><br>      #
    <label for="confirm">Confirm password</label><br>
    <input type="password" name="confirmation" value="{{c
        minlength="8" maxlength="12" required><br>    #6
    <label for="userid">Enter tutor name</label><br>
    <input type="text" name="name" value="{{current_name}
    <label for="imageurl">Enter tutor image url</label><b
    <input type="text" name="imageurl" value="{{current_i
    <label for="profile">Brief tutor profile</label><br>
    <input type="text" name="profile" value="{{current_pr
    <label for="error">                                #7
        <p style="color:red">{{error}}</p>
    </label>
    <br>
    <button type=submit id="button1">Register</button>
</form>
<form action=/signinhome method=GET>
    <button type=submit id="button2">Sign in</button>
</form>
</div>
<p>
<div id="footer">
    (c)Photo by Author
</div>
</p>

</html>

```

Let's now create a file *tutor-styles.css* under *\$PROJECT_ROOT/static* folder and add the following styling to it.

Listing 8.5. *tutor-styles.css*

```
.header {
    padding: 20px;
    text-align: center;
    background: #fad980;
    color: rgb(48, 40, 43);
```

```
    font-size: 30px;
}

.center {
    margin: auto;
    width: 20%;
    min-width: 150px;
    border: 3px solid #ad5921;
    padding: 10px;
}

body, html {
    height: 100%;
    margin: 0;
    font-kerning: normal;
}

h1 {
    text-align: center;
}

p {
    text-align: center;
}

div {
    text-align: center;
}

div {
    background-color: rgba(241, 235, 235, 0.719);
}

body {
    background-image: url('/static/background.jpg');
    background-repeat: no-repeat;
    background-attachment: fixed;
    background-size: cover;
    height: 500px;
}

#button1, #button2 {
    display: inline-block;
}

#footer {
    position: fixed;
```

```

padding: 10px 10px 0px 10px;
bottom: 0;
width: 100%;
/* Height of the footer*/
height: 20px;
}

```

These are pretty standard css constructs, and provided here by way of minimal styling for the landing page showing the tutor registration form. You are encouraged to write your own styling for the page, if you are familiar with css.

Note that the css file refers to a background image */static/background.jpg*. You can find this image uploaded to the git repo for the chapter. Download the file and place it under *\$PROJECT_ROOT/static* folder. Alternatively, you can use your own background image (or none at all).

We're now ready to write the code for the *show_register_form()* handler function.

In *\$PROJECT_ROOT/src/iter5/handler.rs*, update the code as follows:

Listing 8.6. Handler function to show registration form

```

use actix_web::{web, Error, HttpResponse, Result};          #1
use crate::errors::EzyTutorError;                          #1

pub async fn show_register_form(tmpl: web::Data<tera::Tera>) -> R
    let mut ctx = tera::Context::new();                      #3
    ctx.insert("error", "");                                #4
    ctx.insert("current_username", "");
    ctx.insert("current_password", "");
    ctx.insert("current_confirmation", "");
    ctx.insert("current_name", "");
    ctx.insert("current_imageurl", "");
    ctx.insert("current_profile", "");                      #4
    let s = tmpl
        .render("register.html", &ctx)                   #5
        .map_err(|_| EzyTutorError::TeraError("Template error".to
Ok(HttpResponse::Ok().content_type("text/html").body(s))      #
}

```

We can do a quick test now. From `$PROJECT_ROOT` Run the Actix server with the following command, from `$PROJECT_ROOT`:

```
cargo run --bin iter5-ssr
```

Assuming you have followed all the steps described, you should be able to see the landing page showing the registration form, when you visit the following URL from a browser (replace port number with whatever you have configured in the `.env` file):

```
localhost:8080/
```

You have successfully displayed the tutor registration form. It's time to accept user inputs and post the completed form back to the Actix web server. Let's see how that can be done, in the next section.

8.4 Handling registration submission

We've seen how to display the registration form, in the previous section. Go ahead and try to fill out the values. Specifically try the following:

1. Hit the *Register* button without entering any value. You should see the message 'Please fill in this field', or something similar depending upon which browser you use, for all the fields which are marked as *required* in the html template.
2. For input fields where *minlength* or *maxlength* have been specified in the html template, you will see error messages displayed in the browser whenever your input does not meet the criteria.

Note that these are in-browser validations enabled by the HTML specification itself. We have not written any custom code for these validations.

However, these in-browser validations cannot be used to implement more complex validation rules. They have to be implemented in the server-side handler functions. One example of a validation rule in the tutor registration form is that the *password* and the *password confirmation* fields must contain the same value. For this, we will submit the form data to the Actix server and write the validation code in the handler function. (*Note that as mentioned*

earlier, this password check validation can be performed within the browser using jquery or javascript, but we are adopting a pure-Rust approach in this book).

If you recall the registration workflow we saw in the previous section, we also have to perform the following key steps in the handler function:

1. Verify if the *password* and *password confirmation* fields match. If not, return the form back to the user along with a suitable error message. The values the user filled previously should also be returned along with the form, and should not be lost/discard.
2. If the password check is successful, a POST request needs to be made on the backend *tutor web service* to create a new tutor. We'll be using the *awc* crate (from Actix web ecosystem) as the HTTP client to talk to the tutor web service.
3. The web service returns details of the newly created tutor record, which also includes a database-generated *tutor-id*. This *tutor id* represents a unique tutor record in the *tutor web service*. The web application needs to remember this for future use (eg. when requesting the web service for the user profile of the tutor, or to retrieve course list for the tutor). We need to store this information somewhere within the web application.
4. The *username* and *password* entered by the user in the registration form also needs to be recorded within the web application, so it can be used for authenticating the tutor in future.

For storing *tutor-id*, *username* and *password*, we will be using *postgres* as the database. While you can use any database (or even a lighter key value store for this purpose), *postgres* has been chosen as you have already learnt how to use it with Actix in earlier chapters, and this avoids you having to learn how to configure and use yet another datastore with Rust and Actix. If you need a refresher on how to use and configure *postgres* with *sqlx* and Actix, it is recommended that you refer back to *Chapter 4*.

Storing passwords in clear text form in the database is an insecure approach and is highly discouraged for production use. So, we'll use a third-party crate *argon2* for storing hashes of passwords in the database, rather than storing them in clear text form.

Recall that we've already added the `sqlx`, `awc` and `argon2` crates to `Cargo.toml` in the beginning of the chapter. Here is a recap of the three crates that we added.

```
sqlx = {version = "0.3.5", default_features = false, features = [rust-argon2 = "0.8.3"
awc = "2.0.3"
```

Let's now look at the database layer. We need a database only to store registered users with their credentials. We've previously defined the `User` data structure in the `model.rs` file as shown:

```
#[derive(Serialize, Deserialize, Debug, sqlx::FromRow)]
pub struct User {
    pub username: String,
    pub tutor_id: i32,
    pub user_password: String,
}
```

Let's create a table in the database to store user information. In `$PROJECT_ROOT/src/iter5` you've already created a file `dbscripts/user.sql`. Place the following code in this file:

```
drop table if exists ezyweb_user;      #1
create table ezyweb_user              #2
(
    username varchar(20) primary key,
    tutor_id INT,
    user_password CHAR(100) not null
);
```

Login to the `psql` shell prompt. From project root, run the following command:

```
create database __ezytutor_web_ssr__;          #1
create user __ssruser__ with password 'mypassword';  #2
grant all privileges on database ezytutor_web_ssr to ssruser;  #
```

Log out of `psql` and log back in to see if the credentials are working

```
psql -U $DATABASE_USER -d ezytutor_web_ssr -- password
\q
```

Here `$DATABASE_USER` refers to the username created in the database.

Lastly, quit the `psql` shell, and from the project root, run the following command to create the database table. Before that, ensure to set the database user in the environment variable `$DATABASE_USER`, so it becomes convenient for reuse.

```
psql -U $DATABASE_USER -d ezytutor_web_ss < src/iter5/dbscripts/
```

Log back into the `psql` shell, and run the following commands to check if the table has been created correctly.

```
\d+ ezyweb_user
```

You should see the metadata for the table created. If you have any trouble in following these steps related to postgres, refer back to *chapter 4*.

We're now ready to write the database access functions to store and read tutor data. In `$PROJECT_ROOT/src/iter5/dbaccess.rs`, add the following code:

Listing 8.7. Database access function to store and read tutor data

```
use crate::errors::EzyTutorError;                      #1
use crate::model::*;                                    #1
use sqlx::postgres::PgPool;                            #1

//Return result

pub async fn get_user_record(pool: &PgPool, username: String) ->
    // Prepare SQL statement
    let user_row = sqlx::query_as!(
        User,
        "SELECT * FROM ezyweb_user where username = $1",
        username
    )
    .fetch_optional(pool)
    .await?;

    if let Some(user) = user_row {
        Ok(user)
    } else {
        Err(EzyTutorError::NotFound("User name not found".into()))
    }
}
```

```

        }
    }

pub async fn post_new_user(pool: &PgPool, new_user: User) -> Result<User, Error> {
    let user_row = sqlx::query_as!(User, "insert into ezyweb_user (
        new_user.username, new_user.tutor_id, new_user.user_password)
        .fetch_one(pool)
        .await?;

    Ok(user_row)
}

```

Writing such database access functions should be familiar to you by now, as we dealt with them extensively in previous chapters on building the *tutor web service*.

Let's now move on to the handler functions to perform registration.

Now, which handler function should we write to handle registration form submission? You'll recall that when a form is submitted, the browser invokes a POST HTTP request on the `/register` route, and in the `routes` configuration we have specified the handler function as `handle_register()` for this route. Let's head into the `handler.rs` file under `$PROJECT_ROOT/src/iter5`, and update the `handle_register()` function as follows:

Listing 8.8. Function to handle registration form submission

```

use crate::dbaccess::{get_user_record, post_new_user};           #
...                                                               #
use serde_json::json;                                            #

pub async fn handle_register(                                     #
    tmpl: web::Data<tera::Tera>,                                #
    app_state: web::Data<AppState>,                               #
    params: web::Form<TutorRegisterForm>,                         #
) -> Result<HttpResponse, Error> {                            #
    let mut ctx = tera::Context::new();                           #
    let s;                                                       #
    let username = params.username.clone();                      #
    let user = get_user_record(&app_state.db, username.to_string()); #
    let user_not_found: bool = user.is_err();                     #
    //If user is not found in database, proceed to verification o #
    if user_not_found {                                         #
        if params.password != params.confirmation {             #

```

```

        ctx.insert("error", "Passwords do not match");
        ...
        s = tmpl
            .render("register.html", &ctx)
            .map_err(|_| EzyTutorError::TeraError("Template e
} else {
    let new_tutor = json!({
        "tutor_name": ...
    });
    let awc_client = awc::Client::default();                      #7
    let res = awc_client
        .post("http://localhost:3000/tutors/")
        .send_json(&new_tutor)                                     #8
        .await
        .unwrap()                                              #8
        .body()                                                 #9
        .await?;                                               #9
    let tutor_response: TutorResponse = serde_json::from_
    s = format!("Congratulations. ...");                         #11
    // Hash the password
    let salt = b"somerandomsalt";                                #12
    let config = Config::default();                                #12
    let hash =
        argon2::hash_encoded(params.password.clone().as_b
    let user = User {                                         #1
        ...
    };                                                       #2
    let _tutor_created = post_new_user(&app_state.db, use
}
} else {
    ctx.insert("error", "User Id already exists");
    ...
    s = tmpl
        .render("register.html", &ctx)
        ...;      <2,14>
};

Ok(HttpResponse::Ok().content_type("text/html").body(s))
}

```

We are ready to test this. Before that, we have to ensure that the backend tutor web service is running. Go to `ezytutors/tutor-web-service` folder and run the web service with as follows:

```
cargo run --bin iter5
```

Run the web application from `$PROJECT_ROOT` with:

```
cargo run --bin iter5-ssr
```

From a browser, access the URL - `localhost:8080/`. Fill out the form and hit the *Register* button. If all data is entered correctly, you should see a message displayed on the screen:

Congratulations. You have been successfully registered with EzyTu

As a sidenote on the way interaction with the user is handled, the solution presented here is not the best option, for at least two reasons: firstly, in case of error, we end up repeating much code to rebuild the form, and secondly, if the user bookmarks that endpoint thinking it's the registration endpoint, it will actually display a blank page when the bookmark is used. Redirecting to "/" would be a better option. However, this modification is not trivial and is left as an exercise to the reader.

Try registering with the same username again. You should see the registration form populated with the values you entered, along with the following error message.

User Id already exists

Register one more time, but this time ensure that the *password* and *password confirmation* fields don't match. You should once again see the registration form populated with the values you entered, along with the following error message.

Passwords do not match

With this, we conclude the section on registering a tutor.

With this, we conclude this section and chapter. We've seen how to define a template with template variables, display the registration form to the user, perform in-browser and in-handler validations, send an HTTP request from the template, make an HTTP request to a backend web service, and store the user in a local database. We also defined a custom error type to unify error handling. We also learnt how to hash passwords before storing them in a

database for security purposes.

8.5 Summary

- Architecturally, a server-rendered Rust web application consists of *HTML templates* (that are defined and rendered using a template library like *Tera*), *routes* on which HTTP requests arrive, *handler functions* that process the HTTP requests and a *database access layer* that abstracts details of storing and retrieving data.
- A standard HTML form can be used to capture user inputs in an Actix web application. Infusing *Tera* template variables into the HTML form provides a better user experience and feedback to guide the user.
- User input validations in forms can be performed either *within the browser*, or in the server handler function. Normally, simple validations such as field length checks are done using the former, and more complex validations (such as whether the username is already registered) is done in the server handler function. When the user submits the form, a *POST HTTP request* along with the *form data* is sent by the browser to the *Actix web server*, on the specified *route*.
- A *custom error type* can be defined to unify error handling in the web application. In case of errors in the form data entered by the user, the corresponding *form tera template* is re-rendered by the handler function, and sent to the browser, along with a suitable error message.
- Data pertaining to user management (such as username, password) is stored within the web application in a local data store (we have used postgres database in this chapter). The passwords are stored as hashes, and not clear text for security purposes.

In the next chapter, we'll conclude the server-side web application and cover topics including signing-in a user and creating forms for course data maintenance.

See you in the next chapter.

9 Working with forms for course maintenance

This chapter covers

- Designing user authentication
- Setting up the project structure
- Implementing user authentication
- Routing HTTP requests
- Creating a resource with HTTP POST method
- Updating a resource with HTTP PUT method
- Deleting a resource with HTTP DELETE method

In the previous chapter, we looked at *registration* of tutors. You may recall that when a *user* registers as a *tutor*, the information about the tutor is stored across two databases. *Profile details* of the tutor such as *name*, *image* and *area of specialization* are maintained in a database within the backend *tutor web service*. *Registration* details of the user such as *userid* and *password* are stored locally in a database within the *web application*.

In this chapter, we will build on top of the code from the previous chapter. We'll learn to write a Rust front-end web app that *allows* users to sign in to the application, *interact* with a local database, and *communicate* with a backend web service.

Note that the focus in this chapter will not be on writing the *HTML/javascript* user interface for the *web application* (as that is not the focus of this book). Instead, we will focus on writing all the other components that make up a web application in Rust, including *routes*, *request handlers*, and *data models*, and learn how to invoke APIs on the back-end *web service*. In lieu of a *user interface*, we will test the APIs of the web application from a command-line HTTP tool. The task of writing an HTML/javascript-based UI for the web application using *Tera* templates is largely left to the reader as an exercise.

Let's first start with the tutor *sign in* (authentication) functionality.

9.1 Designing user authentication

For tutor *sign in*, we'll accept two fields- *username* and *password*, and use it to authenticate *tutors* to the web application.

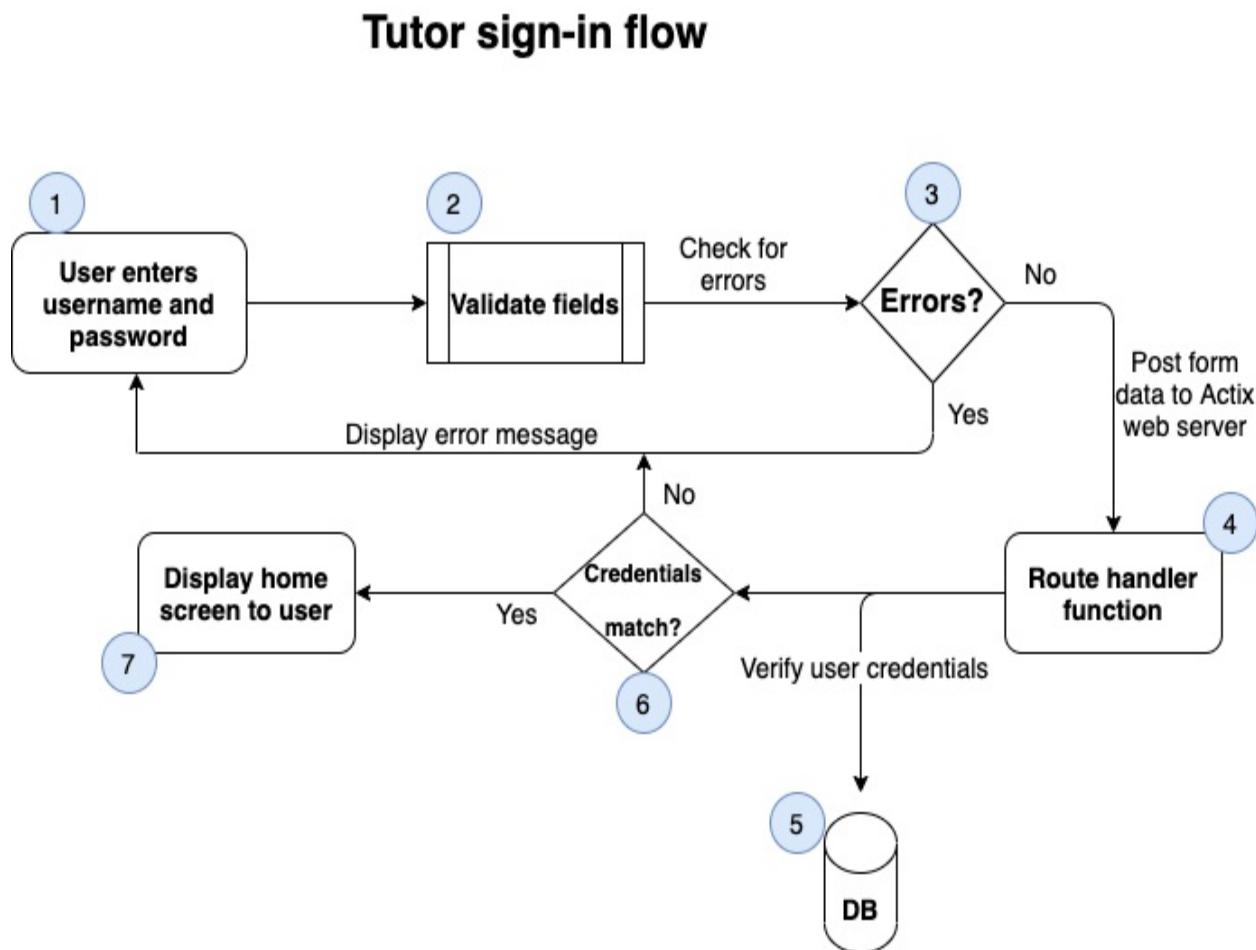
Figure 1 shows the tutor *signin* form.

Figure 9.1. Tutor sign in form

The image shows a mobile application interface for 'Tutor sign in'. The title 'Tutor sign in' is at the top. Below it are two input fields: 'Enter userid' and 'Enter password'. The 'userid' field has a blue border and a key icon with a checkmark to its right. The 'password' field has a white border. At the bottom are 'Sign in' and 'Register' buttons.

Let's now take a look at the workflow for tutor signin in *figure 2*. Note that the term *Actix web server* in *figure 2* refers to the front-end *web application server*, and not the backend *tutor web service*.

Figure 9.2. Tutor signin flow



1. User visits the *landing page URL*. The tutor *signin form* is displayed.
2. Basic validation for username and password is performed within the form itself using HTML features, without having to send requests to the web Actix server.
3. If there are errors in validation, feedback is provided to the user.
4. User submits the *signin form*. A *POST* request is sent to the Actix web server on the *signin route*, which then routes the request to the respective *route handler*.
5. The *route handler function* verifies the username and password, by retrieving the user credentials from the local database.
6. If the authentication is not successful, the *signin form* is displayed back to the user with an appropriate error message. Examples of error messages include incorrect username or password.
7. If the user is authenticated successfully, the user is directed to the home

page of the tutor web application.

Now that we are clear about what we will be developing in this chapter, let's set up the project code structure and basic scaffolding.

9.2 Setting up the project structure

First clone the *ezytutors* repo from chapter8.

Let's then set the *PROJECT_ROOT* environment variable to */path-to-folder/ezytutors/tutor-web-app-ssr*. Henceforth, we'll refer to this folder as *\$PROJECT_ROOT*.

Let's organize the code under project root as follows:

1. Make a copy of the folder *\$PROJECT_ROOT/src/iter5*, and rename it as *\$PROJECT_ROOT/src/iter6*.
2. Make a copy of the folder *\$PROJECT_ROOT/static/iter5*, and rename it as *\$PROJECT_ROOT/static/iter6*. This folder will contain the *html/tera* templates.
3. Make a copy of the file *\$PROJECT_ROOT/src/bin/iter5-ssr.rs*, and rename it to *\$PROJECT_ROOT/src/bin/iter6-ssr.rs*. This file contains the *main()* function that will configure and startup the Actix web server (to serve the web application that we are building). In *iter6-ssr.rs*, replace all references to *iter5* with *iter6*.

Also make sure that the *.env* file in *\$PROJECT_ROOT* is configured correctly for *HOST_PORT* and *DATABASE_URL* environment variables.

We're ready to start coding.

Let's begin with the routes definition in *\$PROJECT_ROOT/src/iter6/routes.rs*.

```
use crate::handler::{handle_register, show_register_form, show_si  
[CA]handle_signin}; #1  
use actix_files as fs;  
use actix_web::web;
```

```

pub fn app_config(config: &mut web::ServiceConfig) {
    config.service(
        web::scope("")
            .service(fs::Files::new("/static", "./static").show_files_1)
            .service(web::resource("/").route(web::get().to(show_register)))
            .service(web::resource("/signinform").route(web::get().to(
                [CA]show_signin_form))) #2
            .service(web::resource("/signin").route(web::post().to(
                [CA]handle_signin))) #3
            .service(web::resource("/register").route(web::post().to(
                [CA]handle_register))),
    );
}

```

With this, we can move on to the model definition in `$PROJECT_ROOT/src/iter6/model.rs`.

Add the `TutorSignInForm` data structure to `model.rs`:

```

// Form to enable tutors to sign in
#[derive(Serialize, Deserialize, Debug)]
pub struct TutorSignInForm { #1
    pub username: String,
    pub password: String,
}

```

With the basic structure of the project setup, we can now start to write code for *signing in* users.

9.3 Implementing user authentication

After defining routes and data model, let's write the handler functions for signing in users, in `$PROJECT_ROOT/src/iter6/handler/auth.rs`.

First, make the following change to the imports:

```
use crate::model::{TutorRegisterForm, TutorResponse, TutorSignInF
```

Add the following handler functions to the same file. Replace references to `iter5` with `iter6` in this file.

```
pub async fn show_signin_form tmpl: web::Data<tera::Tera> ->
```

```

[CA]Result<HttpResponse, Error> { #1
    let mut ctx = tera::Context::new();
    ctx.insert("error", "");
    ctx.insert("current_name", "");
    ctx.insert("current_password", "");
    let s = tmpl
        .render("signin.html", &ctx)
        .map_err(|_| EzyTutorError::TeraError(
            [CA]"Template error".to_string()))?;

    Ok(HttpResponse::Ok().content_type("text/html").body(s))
}

pub async fn handle_signin(                                     #
    tmpl: web::Data<tera::Tera>,
    app_state: web::Data<AppState>,
    params: web::Form<TutorSignInForm>,
) -> Result<HttpResponse, Error> {
    Ok(HttpResponse::Ok().finish())
}

```

Recall that the *show_signin_form* handler function is invoked in response to a request that arrives on route */signinform*, as defined in the routes definition.

Let's design the actual *sign in* html form. This form will be displayed when the user chooses to sign in to the *EzyTutor web application*. Create a new file *signin.html* file under *\$PROJECT_ROOT/static/iter6*, and add the following to it. Note that there should already be another file *register.html* already present in the same folder.

Listing 9.1. Tutor signin form

```

<!doctype html>
<html>

<head>
    <meta charset=utf-8>
    <title>Tutor registration</title>

    <style>                                         #1
        .header {
            padding: 20px;
            text-align: center;
            background: #fad980;
            color: rgb(48, 40, 43);

```

```
        font-size: 30px;
    }

    .center {
        margin: auto;
        width: 20%;
        min-width: 150px;
        border: 3px solid #ad5921;
        padding: 10px;
    }

body,
html {
    height: 100%;
    margin: 0;
    font-kerning: normal;
}

h1 {
    text-align: center;
}

p {
    text-align: center;
}

div {
    text-align: center;
}

div {
    background-color: rgba(241, 235, 235, 0.719);
}

body {
    background-image: url('/static/background.jpg');
    background-repeat: no-repeat;
    background-attachment: fixed;
    background-size: cover;
    height: 500px;
}

#button1,
#button2 {
    display: inline-block;
}
```

```

#footer {
    position: fixed;
    padding: 10px 10px 0px 10px;
    bottom: 0;
    width: 100%;
    /* Height of the footer*/
    height: 20px;
}

</style>
</head>

<body>
    <div class="header">
        <h1>Welcome to EzyTutor</h1>
        <p>Start your own online tutor business in a few minutes</p>
    </div>

    <div class="center">
        <h2>
            Tutor sign in
        </h2>
        <form action=/signin method=POST>      #2

            <label for="userid">Enter username</label><br>
            <input type="text" name="username" autocomplete="user[CA]value="{{current_name}}" minlength="6"
                   maxlength="12" required><br>
            <label for="password">Enter password</label><br>
            <input type="password" name="password"
                   [CA]autocomplete="new-password" value="{{current_pass"
                   minlength="8" maxlength="12" required><br>
            <label for="error">
                <p style="color:red">{{error}}</p>
            </label><br>
            <button type=submit id="button2">Sign in</button>

        </form>
        <form action=/ method=GET>
            <button type=submit id="button2">Register</button>
        </form>
    </div>
    <p>
        <div id="footer">
            (c)Photo by Author
        </div>

```

```
</p>  
</html>
```

Add another file *user.html* to *\$PROJECT_ROOT/static/iter6*. This will be displayed after successful signin by the user.

Listing 9.2. User notification screen

```
<!DOCTYPE html>  
<html>  
  
<head>  
    <meta charset=\"utf-8\" />  
    <title>{{title}}</title>  
</head>  
  
<body>  
    <h1>Hi, {{name}}!</h1>  
    <p>{{message}}</p>  
</body>  
  
</html>
```

Lastly, let's look at the *main()* function in *\$PROJECT_ROOT/src/bin/iter6-ssr.rs*. Modify it to look like below:

Here are the imports:

```
#[path = "../iter6/mod.rs"]  
mod iter6;  
use actix_web::{web, App, HttpServer};  
use actix_web::web::Data;  
use dotenv::dotenv;  
use iter6::{dbaccess, errors, handler, model, routes, state};  
use routes::app_config;  
use sqlx::postgres::PgPool;  
use std::env;  
use tera::Tera;
```

And, this is the *main()* function:

Listing 9.3. main() function

```

#[actix_web::main]
async fn main() -> std::io::Result<()> {
    dotenv().ok();
    //Start HTTP server
    let host_port = env::var("HOST_PORT").expect(
        [CA]"HOST:PORT address is not set in .env file");
    println!("Listening on: {}", &host_port);
    let database_url = env::var("DATABASE_URL").expect(
        [CA]"DATABASE_URL is not set in .env file");
    let db_pool = PgPool::connect(&database_url).await.unwrap();
    // Construct App State
    let shared_data = web::Data::new(state::AppState { db: db_poo
HttpServer::new(move || {
    let tera = Tera::new(concat!(env!("CARGO_MANIFEST_DIR"),
[CA]"/static/iter6/**/*")).unwrap();

    App::new()
        .app_data(Data::new(tera))
        .app_data(shared_data.clone())
        .configure(app_config)
})
    .bind(&host_port)?
    .run()
    .await
}

```

We can test now. Run the following command from `$PROJECT_ROOT`.

```
cargo run --bin iter6-ssr
```

Note: If you get the error: *no implementation for `u32 - usize`:*

Run the following:

```
cargo update -p lexical-core
```

From a browser, access the following route:

```
localhost:8080/signinform
```

You should be able to see the signin form. You can also invoke the signing form by accessing the index route `/`, which shows the registration form, and by using the button shown to switch to the *signin* form.

Once you have this working, you are ready to implement the logic for signing in the user. Add the following to `$PROJECT_ROOT/src/iter6/handler.rs`. Don't forget to remove the placeholder function with the same name created earlier.

Listing 9.4. Handler function for signin

```
pub async fn handle_signin(
    tmpl: web::Data<tera::Tera>,
    app_state: web::Data<AppState>,
    params: web::Form<TutorSigninForm>,
) -> Result<HttpResponse, Error> {
    let mut ctx = tera::Context::new();
    let s;
    let username = params.username.clone();
    let user = get_user_record(&app_state.db, username.to_string());
    if let Ok(user) = user {
        let does_password_match = argon2::verify_encoded(
            &user.user_password.trim(),
            params.password.clone().as_bytes(),
        )
        .unwrap();
        if !does_password_match {
            ctx.insert("error", "Invalid login");
            ctx.insert("current_name", &params.username);
            ctx.insert("current_password", &params.password);
            s = tmpl
                .render("signin.html", &ctx)
                .map_err(|_| EzyTutorError::TeraError(
                    [CA]"Template error".to_string()))?;
        } else {
            ctx.insert("name", &params.username);
            ctx.insert("title", &"Signin confirmation!".to_owned());
            ctx.insert(
                "message",
                &"You have successfully logged in to EzyTutor!".t
            );
            s = tmpl
                .render("user.html", &ctx)
                .map_err(|_| EzyTutorError::TeraError(
                    [CA]"Template error".to_string()))?;
        }
    } else {
        ctx.insert("error", "User id not found");
        ctx.insert("current_name", &params.username);
        ctx.insert("current_password", &params.password);
    }
}
```

```

        s = tmpl
            .render("signin.html", &ctx)
            .map_err(|_| EzyTutorError::TeraError(
                [CA]"Template error".to_string()))?;
    };

    Ok(HttpResponse::Ok().content_type("text/html").body(s))
}

```

Let's test the `signin` function now. Run the following command from `$PROJECT_ROOT`.

```
cargo run --bin iter6-ssr
```

From a browser, access the following route:

```
localhost:8080/signinform
```

Enter the correct username and password. You should see the confirmation message.

Load the *signin form* once again, and this time enter a wrong password for a valid username. Verify that you get the error message.

Try entering the form the third time, this time with an invalid user name. Again, you should see an error message.

With this, we conclude this section. We've so far seen how to define templates using *Tera* template library to generate dynamic web pages, and to display the *registration* and *sign in* forms to the user. We've also implemented the code to *register* and *sign in* a user, and handle errors in user inputs. We also defined a custom error type to unify error handling.

Let's now move on to managing course details.

9.4 Routing HTTP requests

In this section, we'll add the ability for a tutor to maintain courses.

We currently have all handler functions in a single file. We'll now have to

add handlers for course maintenance also. So, let's first organize handler functions into its own module, that gives the ability to split the handler functions across multiple source files.

Start by creating a new *handler* folder under `$PROJECT_ROOT/src/iter6`.

Move `$PROJECT_ROOT/src/iter6/handler.rs` into `$PROJECT_ROOT/src/iter6/handler` and rename it as `auth.rs`, as this deals with registration and login functionality. (i.e. `mv $PROJECT_ROOT/src/iter6/handler.rs $PROJECT_ROOT/src/iter6/handler/auth.rs` in linux).

Create new files `course.rs` and `mod.rs` under `$PROJECT_ROOT/src/iter6/handler` folder. In `mod.rs` add the following code to structure the files in the *handler* folder and export them as a Rust module.

```
pub mod auth;    #1
pub mod course; #2
```

Modify `$PROJECT_ROOT/src/iter6/routes.rs` as shown:

Listing 9.5. Adding routes for course maintenance

```
use crate::handler::auth::{handle_register, handle_signin,
[CA]show_register_form, show_signin_form}; #1
use crate::handler::course::{handle_delete_course, handle_insert_
[CA]handle_update_course}; #2

use actix_files as fs;
use actix_web::web;

pub fn app_config(config: &mut web::ServiceConfig) {                      #
    config.service(
        web::scope("")
            .service(fs::Files::new("/static", "./static").show_files_1
            .service(web::resource("/").route(web::get().to(show_register
            .service(web::resource("/signinform").route(web::get().to(
                [CA]show_signin_form)))
            .service(web::resource("/signin").route(web::post().to(
                [CA]handle_signin)))
            .service(web::resource("/register").route(web::post().to(
```

```

        [CA]handle_register))),
    );
}

pub fn course_config(config: &mut web::ServiceConfig) {
    config.service(
        web::scope("/courses")
            .service(web::resource("new/{tutor_id}").route(web::post() .
                [CA]handle_insert_course))) #6
            .service(
                web::resource("{tutor_id}/{course_id}").route(web::put() .
                    [CA]handle_update_course)),
        )
        .service(
            web::resource("delete/{tutor_id}/{course_id}")
                .route(web::delete().to(handle_delete_course)),
        ),
    );
}

```

Note that where we have specified the `{tutor_id}` and `{course_id}` as path parameters, they can be extracted from the request's path with help of *extractors* provided by the Actix web framework.

Also make sure to add the new course maintenance routes in `$PROJECT_ROOT/bin/iter6-ssr.rs` as shown:

Make the following change to import statement:app-name:

```
use routes::{app_config, course_config};
```

In the `main()` function, make the change to add `course_config` routes.

```

HttpServer::new(move || {
    let tera = Tera::new(concat!(env!("CARGO_MANIFEST_DIR"),
        [CA]"/static/iter6/**/*")).unwrap();

    App::new()
        .app_data(Data::new(tera))
        .app_data(shared_data.clone())
        .configure(course_config) #1
        .configure(app_config) #2
})
.bind(&host_port)?
.run()

```

```
.await
```

Next, let's for now add the placeholder handler functions for course maintenance in `$PROJECT_ROOT/src/iter6/handler/course.rs`. We'll write the actual logic to call the backend web service, a little later.

Listing 9.6. Placeholders for course maintenance handler functions

```
use actix_web::{web, Error, HttpResponse, Result};
use crate::state::AppState;

pub async fn handle_insert_course(
    _tmpl: web::Data<tera::Tera>,
    _app_state: web::Data<AppState>,
) -> Result<HttpResponse, Error> {
    println!("Got insert request");
    Ok(HttpResponse::Ok().body("Got insert request"))
}

pub async fn handle_update_course(
    _tmpl: web::Data<tera::Tera>,
    _app_state: web::Data<AppState>,
) -> Result<HttpResponse, Error> {
    Ok(HttpResponse::Ok().body("Got update request"))
}

pub async fn handle_delete_course(
    _tmpl: web::Data<tera::Tera>,
    _app_state: web::Data<AppState>,
) -> Result<HttpResponse, Error> {
    Ok(HttpResponse::Ok().body("Got delete request"))
}
```

As you will note, the handler functions do nothing for now, except to return a message. We will implement the intended handler functionality later in this chapter.

Note the use of underscore (`_`) before the variable names. This is because, we are not going to be using these parameters within the body of the handler function yet, and so adding an underscore before the variable names will prevent compiler warnings.

Let's do a quick test of these four routes:

Run the server with:

```
cargo run --bin iter6-ssr
```

To test the *POST*, *PUT* and *DELETE* requests, try the following from the command line:

```
curl -H "Content-Type: application/json" -X POST -d '{}' [CA]localhost:8080/courses/new/1  
curl -H "Content-Type: application/json" -X PUT -d '{}' [CA]localhost:8080/courses/1/2  
curl -H "Content-Type: application/json" -X DELETE -d '{}' [CA]localhost:8080/courses/delete/1/2
```

You should see the following messages returned from the server, corresponding to the three HTTP requests shown above:

```
Got insert request  
Got update request  
Got delete request
```

We've now verified that the routes have been established correctly, and the HTTP requests are being routed to the correct handler functions. In the next section, let's implement the actual logic for adding a course for a tutor in the handler function.

9.5 Creating a resource with HTTP POST method

In this section, we'll add a new course for a given tutor, by sending an API request to the backend tutor web service.

Go to the code repo for Chapter 6 (i.e., */path-to-chapter4-folder/ezytutors/tutor-db*), and start the *tutor web service* with the following command:

```
cargo run --bin iter5
```

The tutor *web service* should now be ready to receive requests from the tutor *web application*. Let's now write the code for the *course handler* in the web application, in *\$PROJECT_ROOT/src/iter6/handler/course.rs*.

Modify the `$PROJECT_ROOT/src/iter6/model.rs` to add the following:

Listing 9.7. Data model changes for course maintenance

```
#[derive(Deserialize, Debug, Clone)]
pub struct NewCourse {                                     #1
    pub course_name: String,
    pub course_description: String,
    pub course_format: String,
    pub course_duration: String,
    pub course_structure: Option<String>,
    pub course_price: Option<i32>,
    pub course_language: Option<String>,
    pub course_level: Option<String>,
}

#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct NewCourseResponse {                           #2
    pub course_id: i32,
    pub tutor_id: i32,
    pub course_name: String,
    pub course_description: String,
    pub course_format: String,
    pub course_structure: Option<String>,
    pub course_duration: String,
    pub course_price: Option<i32>,
    pub course_language: Option<String>,
    pub course_level: Option<String>,
    pub posted_time: String,
}

impl From<web::Json<NewCourseResponse>> for NewCourseResponse {
    fn from(new_course: web::Json<NewCourseResponse>) -> Self {
        NewCourseResponse {
            tutor_id: new_course.tutor_id,
            course_id: new_course.course_id,
            course_name: new_course.course_name.clone(),
            course_description: new_course.course_description.clone(),
            course_format: new_course.course_format.clone(),
            course_structure: new_course.course_structure.clone(),
            course_duration: new_course.course_duration.clone(),
            course_price: new_course.course_price,
            course_language: new_course.course_language.clone(),
            course_level: new_course.course_level.clone(),
            posted_time: new_course.posted_time.clone(),
        }
    }
}
```

```
        }
    }
```

Also make sure to add the following module import, which is required by the *From* trait implementation.

```
use actix_web::web;
```

Next, let's re-write the handler function to create a new course. In *\$PROJECT_ROOT/src/iter6/handler/course.rs*, add the following module imports:

```
use actix_web::{web, Error, HttpResponse, Result}; #1
use crate::state::AppState;
use crate::model::{NewCourse, NewCourseResponse, UpdateCourse, Up
use serde_json::json; #3

use crate::state::AppState;
```

Then modify the *handle_insert_course* handler function as shown:

Listing 9.8. Handler function for inserting a new course

```
pub async fn handle_insert_course( #1
    _tmpl: web::Data<tera::Tera>, #2
    _app_state: web::Data<AppState>, #3
    path: web::Path<i32>,
    params: web::Json<NewCourse>, #4
) -> Result<HttpResponse, Error> {
    let tutor_id = path.into_inner(); #5
    let new_course = json!({
        "tutor_id": tutor_id, #6
        "course_name": &params.course_name,
        "course_description": &params.course_description,
        "course_format": &params.course_format,
        "course_structure": &params.course_structure,
        "course_duration": &params.course_duration,
        "course_price": &params.course_price,
        "course_language": &params.course_language,
        "course_level": &params.course_level
    });
    let awc_client = awc::Client::default(); #7
    let res = awc_client #8
        .post("http://localhost:3000/courses/")
```

```

    .send_json(&new_course)
    .await
    .unwrap()
    .body()
    .await?;
println!("Finished call: {:?}", res);
let course_response: NewCourseResponse = serde_json::from_str
[CA]&std::str::from_utf8(&res)?;      #9
Ok(HttpResponse::Ok().json(course_response))           #
}

```

Build and run the Web ssr client from the `$PROJECT_ROOT` as shown:

```
cargo run --bin iter6-ssr
```

Let's test the new course creation with a curl request. Ensure that the *tutor web service* is running. From another terminal, run the following command:

```
curl -X POST localhost:8080/courses/new/1 -d '{"course_name": "Rust development", "course_description": "Teaches how to write web Rust", "course_format": "Video", "course_duration": "3 hours", "course_price": 100}' -H "Content-Type: application/json"
```

Verify if the new course has been added by running a GET request on the *tutor web service*:

```
curl localhost:3000/courses/1
```

You should see the new course in the list of courses retrieved for *tutor-id* = 1.

In the next section, we'll write the handler function to update a course.

9.6 Updating a resource with HTTP PUT method

Let's write the data structure for updating a course in `$PROJECT_ROOT/src/iter6/model.rs` file.

Listing 9.9. Data model changes for updating courses

```
// Update course
#[derive(Deserialize, Serialize, Debug, Clone)] #1
pub struct UpdateCourse {
```

```

    pub course_name: Option<String>,
    pub course_description: Option<String>,
    pub course_format: Option<String>,
    pub course_duration: Option<String>,
    pub course_structure: Option<String>,
    pub course_price: Option<i32>,
    pub course_language: Option<String>,
    pub course_level: Option<String>,
}
#[derive(Deserialize, Serialize, Debug, Clone)]
pub struct UpdateCourseResponse { #2
    pub course_id: i32,
    pub tutor_id: i32,
    pub course_name: String,
    pub course_description: String,
    pub course_format: String,
    pub course_structure: String,
    pub course_duration: String,
    pub course_price: i32,
    pub course_language: String,
    pub course_level: String,
    pub posted_time: String,
}
impl From<web::Json<UpdateCourseResponse>> for UpdateCourseResponse
{
    fn from(new_course: web::Json<UpdateCourseResponse>) -> Self
        UpdateCourseResponse {
            tutor_id: new_course.tutor_id,
            course_id: new_course.course_id,
            course_name: new_course.course_name.clone(),
            course_description: new_course.course_description.clone(),
            course_format: new_course.course_format.clone(),
            course_structure: new_course.course_structure.clone(),
            course_duration: new_course.course_duration.clone(),
            course_price: new_course.course_price,
            course_language: new_course.course_language.clone(),
            course_level: new_course.course_level.clone(),
            posted_time: new_course.posted_time.clone(),
        }
    }
}

```

You'll also notice that we have defined similar data structures for creating a course (*NewCourse*, *NewCourseResponse*) and for updating a course (*UpdateCourse*, *UpdateCourseResponse*). Is it possible to optimize by

reusing the same structs for both create and update operations? Some optimisation may be possible in a real-project scenario. However for the sake of writing this example code, we have assumed that for creating a new course, the set of mandatory fields needed are different from that needed to update a course (where there is no mandatory field). Also, separating data structs for *create* and *update* operations makes it easier to understand, while learning.

Next, let's rewrite the handler function to update course details in `$PROJECT_ROOT/src/iter6/handler/course.rs`.

Listing 9.10. Handler function for updating a course

```
pub async fn handle_update_course(
    _tmpl: web::Data<tera::Tera>,
    _app_state: web::Data<AppState>,
    web::Path((tutor_id, course_id)): web::Path<(i32, i32)>,
    params: web::Json<UpdateCourse>,
) -> Result<HttpResponse, Error> {
    let update_course = json!({                                     #1
        "course_name": &params.course_name,
        "course_description": &params.course_description,
        "course_format": &params.course_format,
        "course_duration": &params.course_duration,
        "course_structure": &params.course_structure,
        "course_price": &params.course_price,
        "course_language": &params.course_language,
        "course_level": &params.course_level,
    });
    let awc_client = awc::Client::default();                      #2
    let update_url = format!("http://localhost:3000/courses/{}/{}"
        [CA]tutor_id, course_id);      #3
    let res = awc_client                                         #4
        .put(update_url)
        .send_json(&update_course)
        .await
        .unwrap()
        .body()
        .await?;
    let course_response: UpdateCourseResponse = serde_json::from_
        [CA]&std::str::from_utf8(&res)?; #5
    Ok(HttpResponse::Ok().json(course_response))
}
```

```
}
```

Make sure to import the update-related structs as shown:

```
use crate::model::{NewCourse, NewCourseResponse, UpdateCourse, Up
```

Build and run the Web ssr client from the \$PROJECT_ROOT as shown:

```
cargo run --bin iter6-ssr
```

Let's test with a curl request to update the course we previously created. Ensure that the *tutor web service* is running. From a new terminal, run the following command. Replace the *tutor-id* and *course-id* with those of the new course that you previously created.

```
curl -X PUT -d '{"course_name": "Rust advanced web development",  
[CA]"course_description": "Teaches how to write advanced web apps  
[CA]"course_format": "Video", "course_duration": "4 hours",  
[CA]"course_price": 100}' localhost:8080/courses/1/27 -H  
[CA]"Content-Type: application/json"
```

Verify if the course details have been updated by running a GET request on the *tutor web service*:

```
curl localhost:3000/courses/1
```

Note: Replace *course_id: 1* with the correct value for the *tutor_id* for which you updated the course.

You should see the updated course details reflected.

Let's move on to deleting a course.

9.7 Deleting a resource with HTTP DELETE method

Let's update the handler function to delete a course in \$PROJECT_ROOT/src/iter6/handler/course.rs.

Listing 9.11. Handler function for deleting a course

```
pub async fn handle_delete_course(
    _tmpl: web::Data<tera::Tera>,
    _app_state: web::Data<AppState>,
    path: web::Path<(i32, i32)>,      #1
) -> Result<HttpResponse, Error> {
    let (tutor_id, course_id) = path.into_inner();
    let awc_client = awc::Client::default();          #
    let delete_url = format!("http://localhost:3000/courses/{}/{}"
        [CA]tutor_id, course_id);      #3
    let _res = awc_client.delete(delete_url).send().await.unwrap(
        Ok::(Ok()).body("Course deleted"))
}
```

Build and run the *tutor web app* from the \$PROJECT_ROOT as shown:

```
cargo run --bin iter6-ssr
```

Run the delete request as shown:

```
curl -X DELETE localhost:8080/courses/delete/1/19
```

Replace *tutor_id* and *course_id* with your own.

Verify if the course has been deleted by running a query on the *tutor web service*.

```
curl localhost:3000/courses/1
```

Replace the *tutor_id* with your own. You should see that the course has been deleted in the *_tutor web service*.

With this, we have seen how to add, update and delete a course from the web client front-end written in Rust.

As an exercise, readers can do the following additional tasks:

1. Implement a new route to retrieve the list of courses for a tutor
2. Create HTML/Tera templates for creating, updating and deleting a course
3. Add additional error handling for cases with invalid user inputs.

9.8 Summary

- In this chapter, we learnt how to structure and write a web application project in Rust that talks to a backend web service.
- We designed and implemented the user authentication functionality that allows the user to enter the credentials in an HTML form, and then stores them in a local database. Handling of errors in user inputs was also covered.
- We discussed how to structure the project and modularize the code for a web front-end application that includes *HTTP request handlers, database interaction logic, data model and web UI/Html templates*.
- We wrote code to *create, update and delete* specific data in the database in response to HTTP *POST, PUT* and *DELETE* method requests. We also understood how to extract parameters sent as part of the HTTP requests.
- We learnt how to construct HTTP requests to invoke *APIs* on a *backend web service*, and to interpret the responses received, including data serialization and deserialization.
- In summary, you have learnt how to build a web application in Rust that can communicate with a backend web service, interact with a local database and perform basic create, update and delete operations on data in response to incoming HTTP requests.

With this, we come to the conclusion of this chapter and also this section on Rust web application development.

In the next chapter, we'll take a look at an advanced topic relating to asynchronous servers in Rust.

See you in the next chapter.

10 Understanding Async Rust

This chapter covers

- Introduction to Async programming concepts
- Writing concurrent programs
- Diving deeper into async Rust
- Understanding futures
- Implementing a custom future

In the previous chapters, we covered building a *web service* and a *web application* using Rust. To build these, we've used the *Actix web framework* for handling the network communications. We've mostly submitted HTTP requests to the Actix web server from a single browser window or from a command-line terminal. But have you thought about what happens when tens or hundreds of users send requests concurrently for registering tutors or courses? Or more broadly, how do modern web servers handle tens of thousands of concurrent requests? Read on, to find out.

In this chapter, we will take a detour from building the web application, and look under the hood to understand what is asynchronous Rust, what is the need to use it, and how it works in practice. By the end of this chapter you'll have a better understanding of the magic that Actix (and other similar modern web frameworks) perform to handle heavy concurrent loads, while delivering swift responses to user requests.

Note that this chapter is intended as an advanced topic, aimed at those who want to get into the details of asynchronous programming in Rust. However, it is not necessary to complete this chapter to do web programming in Rust. As a reader, you can choose to skip this chapter and come back to it at a later stage when you are ready for an async deep-dive.

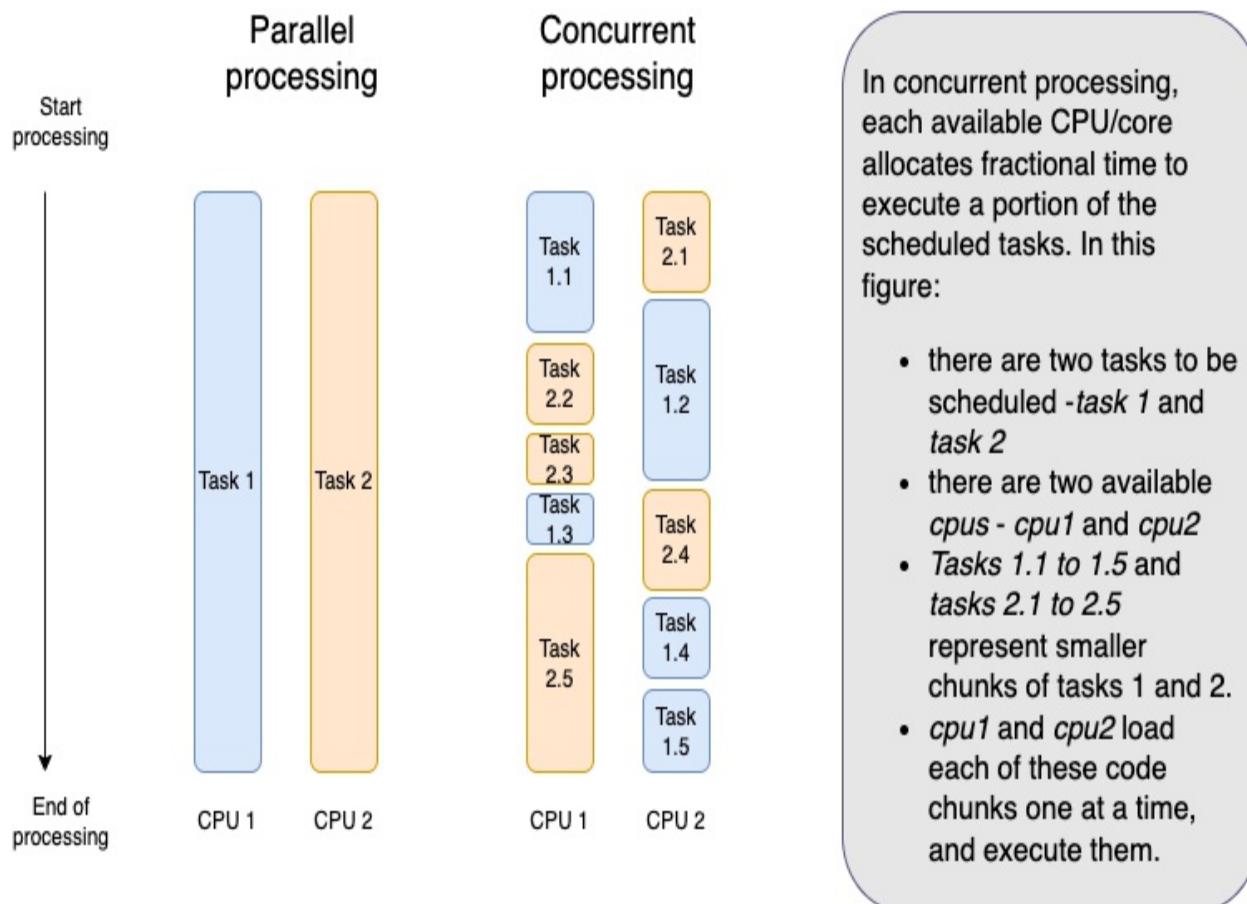
Let's now get started with a few basic concepts of concurrent programming.

10.1 Introduction to Async programming concepts

In computer science, concurrency is the ability of different parts of a program to be executed out-of-order or at the same time simultaneously, without affecting the final outcome.

Strictly speaking, executing parts of a program out-of-order is *concurrency*, while executing multiple tasks simultaneously is *parallelism*. But in practice, both *concurrency* and *parallelism* are used in conjunction to achieve the overall outcome of processing multiple requests arriving at the same time in an efficient and safe manner. Figure 10.1 illustrates this difference. However, for this chapter, let's use the term *concurrency* to broadly refer to both of these aspects.

Figure 10.1. Concurrency vs Parallelism



Now you may wonder, why would one want to execute parts of a program out-of-order? After all, programs are supposed to execute from top to bottom, statement by statement, right?

There are two primary drivers for doing concurrent programming - one from the *demand* side and another from the *supply* side.

On the *demand* side (as in user demand), the expectation for programs to run faster drives software devs to consider *concurrent programming* techniques.

On the *supply* side (as in hardware supply), the availability of multiple CPUs (and/or multiple cores in CPUs) on computers (even the ones that are sold to end users like you and me, and not just the high-end servers in data centers) creates an opportunity for software developers to write programs that can take advantage of multiple cores/processors available, in order to make the overall execution faster and efficient.

But designing and coding concurrent programs is a complex task. It starts with determining what tasks to perform concurrently. How do the developers determine which parts of code can be executed concurrently?

Let's go back to Figure 10.1. It shows two tasks - task 1 and task 2 to be executed. Let's assume here that tasks 1 and 2 are two functions in a Rust program. The easiest way to visualize is to schedule task1 on cpu1 and task2 on cpu2. This is shown under parallel processing. But is this the most efficient model for utilizing the available CPU time?

It may not be. To understand this better, let's classify all processing performed by software programs broadly into two categories: *CPU-intensive* tasks and *I/O-intensive* tasks, even though most code in the real-world involves a mix of both. Examples of *CPU-intensive* tasks are genome sequencing, video encoding, graphics processing and computing cryptographic proofs in a blockchain. Examples of *I/O-intensive* tasks are accessing data from file systems or databases, and processing network TCP/HTTP requests.

In CPU-intensive tasks, most of the work involves accessing data in memory, loading the program instructions and data on the stack, and executing them. What kind of concurrency is possible here? Let's take a simple example of a program that takes a list of numbers and computes the square root of each number. The programmer can write a single function that

- takes a reference to a list of numbers loaded into memory,
- iterates through the list in a sequence,
- computes the square root for each number and
- writes the result back to memory.

This would be an example of sequential processing. In a computer where there are multiple processors/cores, the programmer also has the opportunity to structure the program in such a way that each number is read from memory and sent for square-root processing to the next available CPU/core, as each number can be processed independent of the other. While, this is a trivial example, it gives an idea of the type of opportunity available for programmers to utilize multiple processors/cores in complex *computation-intensive* tasks.

Let's next look at where the opportunity is for concurrency in *I/O-intensive* tasks. Here, let's take the familiar example of HTTP request processing in web services and applications, which is generally more *I/O-intensive* than *CPU-intensive*.

In web applications, data is stored in databases, and all *Create, Read, Update, Delete* operations, corresponding to **HTTP POST, GET, PUT** and **DELETE** requests respectively, require the web application to transfer data to and from the database. This requires the processor (CPU) to wait for the data to be read or written to disk. And in spite of advances in disk technologies, disk access is slow (in the range of milliseconds as opposed to memory access which is in nanoseconds). So, if the application is trying to retrieve 10,000 user records from a Postgres database, it makes calls to the operating system for disk access, and the CPU 'waits' during this time. Now, what options does the programmer have when a part of her code makes the processor wait? The answer is to have the processor perform another task. This is an example of an opportunity available to programmers to design concurrent programs.

Another source of 'delays' or 'waiting' in web applications is network request handling. The HTTP model is quite simple. The client establishes a connection to the remote server and issues a request (sent as an HTTP request message). The server then processes the request, issues a response and closes the connection. (Note: HTTP/2 has brought some improvements to minimize

the number of request-response cycles and handshakes. For more details on HTTP/2, here is a book reference: www.manning.com/books/http2-in-action). The challenge arises when a new request arrives while the processor is still serving the previous request. For example, a GET request arrives to retrieve a set of courses for *Tutor 1*, and while this is still being processed, a new request arrives to POST a new course from *Tutor 2*. Should the second request wait in queue until the first request is fully processed? Or can we schedule the second request on the next available core/processor? This is when we start to appreciate the need for concurrent programming.

We have so far seen examples of opportunities available to programmers to use concurrent programming techniques both in *computation-intensive tasks* and *I/O-intensive tasks*. Let's now look at the tools available to programmers to write concurrent programs.

Figure 10.2. Synchronous, asynchronous and multi-threading

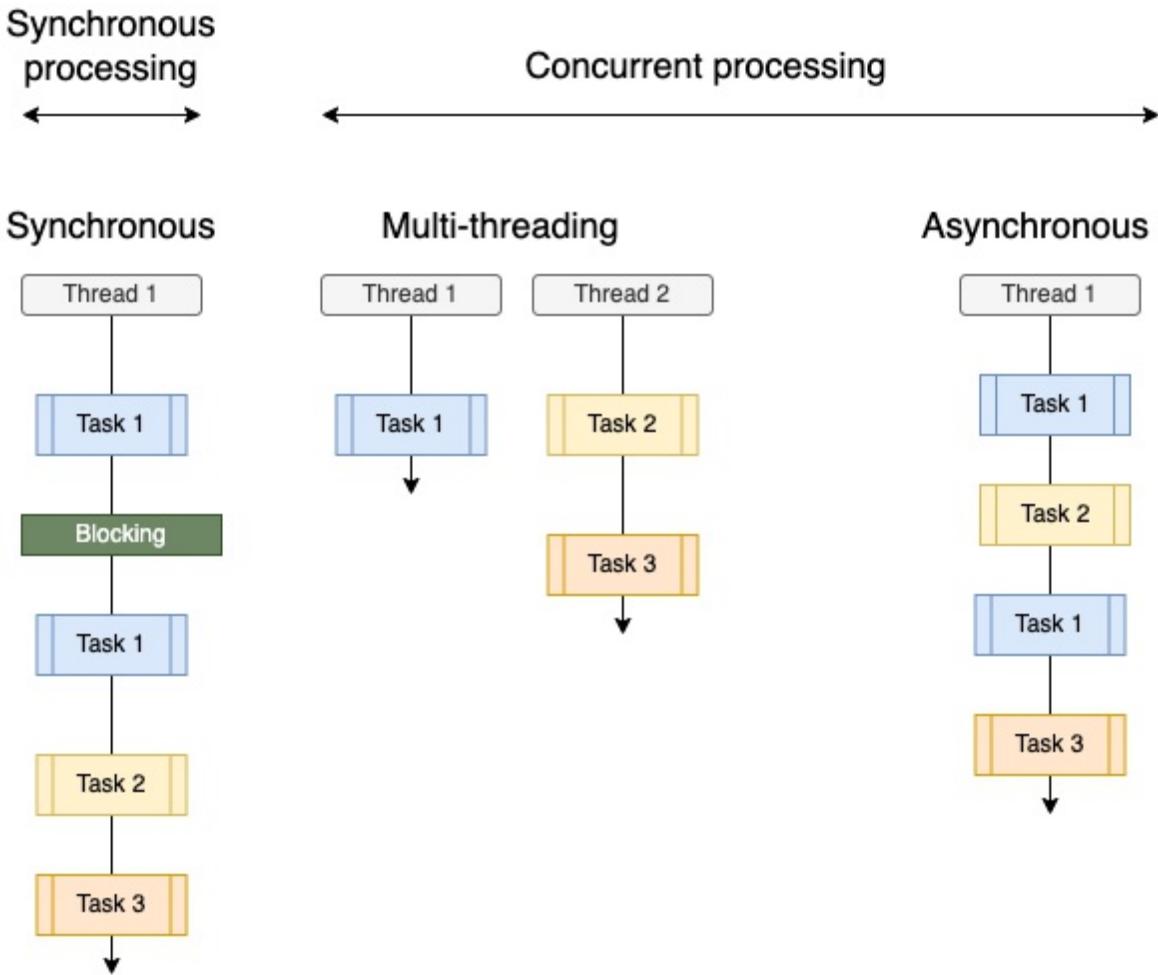


Figure 10.2 shows the various options available to programmers to structure their code for execution on the CPU(s). To be more specific, it highlights the differences between *synchronous* processing and the two modes of *concurrent* processing - *multi-threading* and *async* processing. It illustrates the differences using an example of a case where there are three tasks to be executed - *task 1*, *task 2* and *task 3*.

Let's also assume *task 1* to contain three parts:

- part-1: processing of input data,
- part-2: a blocking operation, and then
- part-3: packaging the data to be returned from the task

Note the blocking operation. This means that the current thread of execution is blocked waiting for some external operation to complete, e.g., reading from

a large file or database.

Let's now look at how to handle these tasks in three different programming modes - **synchronous** processing, **multi-threaded** processing and **async** processing.

In the case of **synchronous** processing, the processor completes *part-1*, waits for the result of the blocking operation, and then proceeds to execute part-3 of the task.

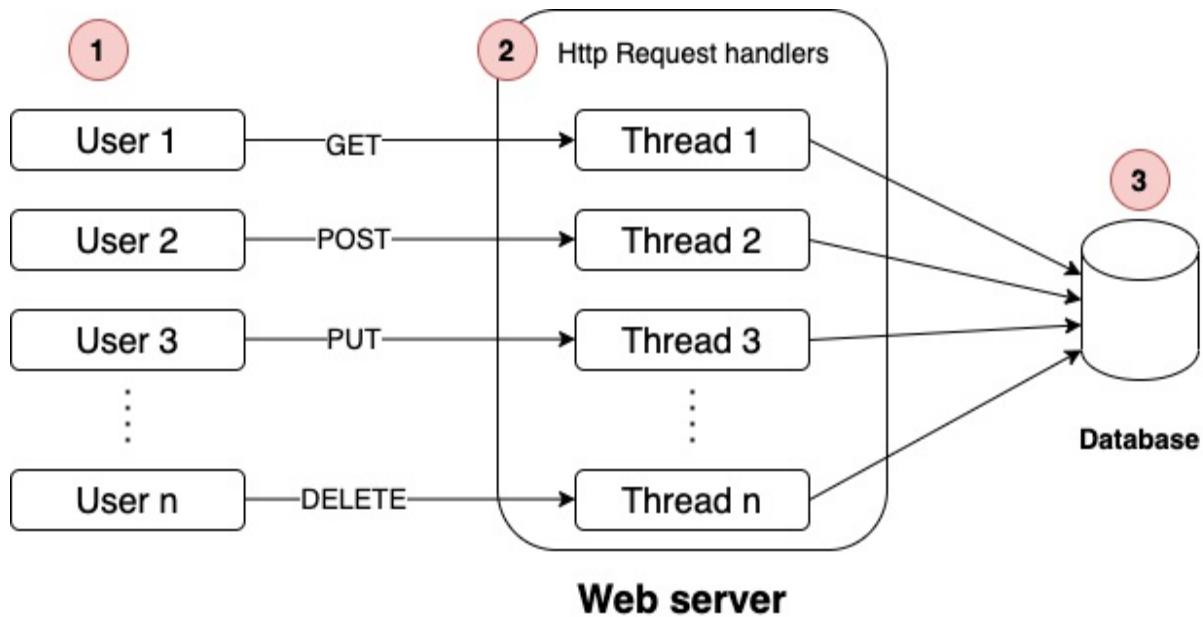
If the same task were to be executed in **multi-threaded** mode, *task 1* that contains the blocking operation can be spawned off on a separate operating system thread, while the processor can execute other tasks on another thread.

If **async** processing is used, an async runtime (such as *Tokio*) manages scheduling of tasks on the processor. In this case, it executes *task 1* until the point when it blocks waiting for I/O. At this point, the *async runtime* schedules the second task. When the blocking operation completes on the first task, it is then scheduled for execution back on the processor.

At a high level, this is how synchronous processing differs from the two modes of concurrent processing. It is left to the programmer to determine which is the best approach for the particular use case and computation involved.

Let's now go to the second example of a web server receiving multiple simultaneous network requests, and see how the two types of concurrent processing techniques can be applied here.

Figure 10.3. Multi-threading in HTTP request processing



- 1 A number of simultaneous HTTP requests are sent from the users to the web server
- 2 For each incoming request, a separate operating system thread is created to handle the incoming request
- 3 The operating system schedules each thread for execution on the processor/core , which make database requests and send back responses to the users.

The first approach to concurrency involves using native operating system threads as shown in figure 10.3, i.e. start a new thread within the web server process to handle each incoming request. The Rust standard library provides good built-in support for multi-threading with the `std::thread` module. In this model, we are distributing the program (web server) computation on to multiple threads. This can improve performance because threads can run simultaneously. However, it's not as simple as that. Multi-threading adds a new layer of complexity including

- *unpredictability* about order of execution of threads,
- *deadlocks* where multiple threads are trying to access the same piece of data in memory, and
- *race conditions* (where for example one thread may have read a piece of data from memory and is performing some computation with it, while another thread updates the value in the meantime).

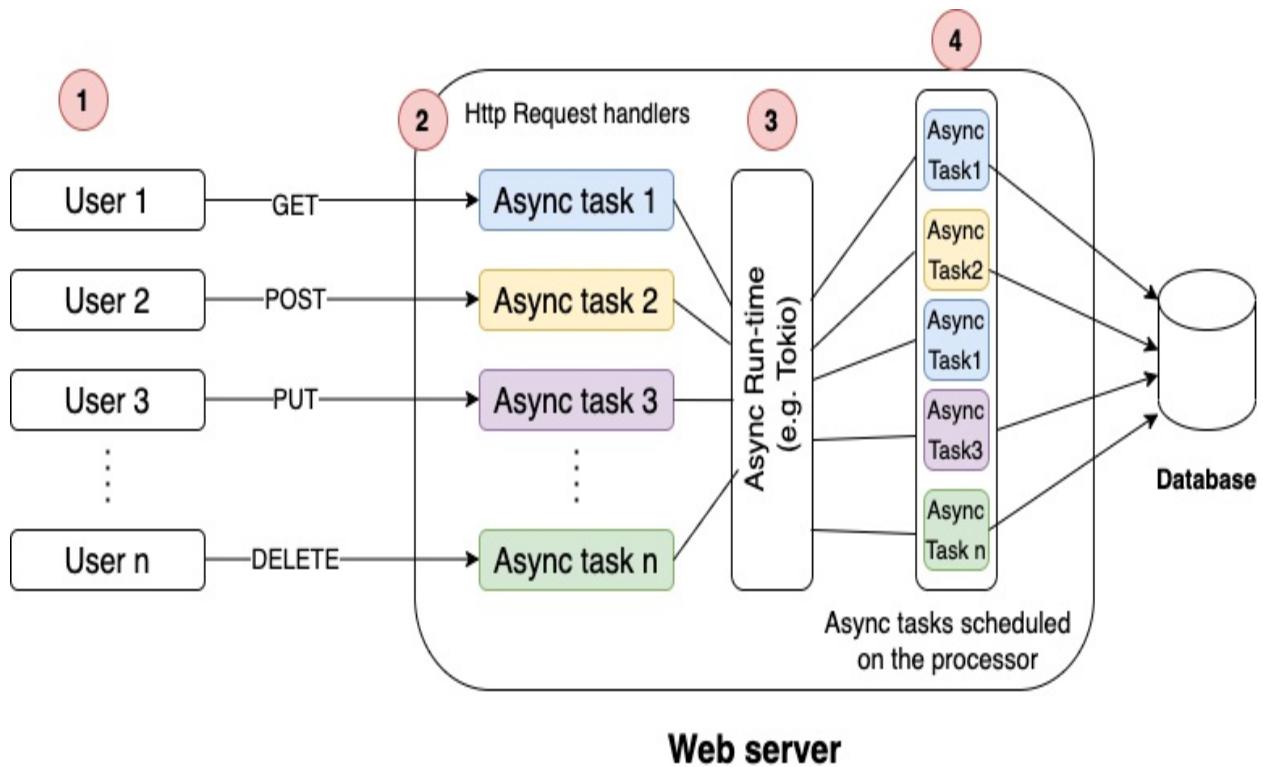
Writing multi-threaded programs requires careful design compared to single-

threaded programs.

There is another challenge in *multi-threading*, which is to do with the type of *threading model* implemented by the programming language. There are two types of threading models: *1:1 thread model* where there is a single operating system thread per language thread, and *M:N model* where there are M green (quasi) threads per N operating system threads. The Rust standard library implements the 1:1 thread model. But this does not mean that we can create an endless number of threads corresponding to new network requests, as generally each operating system has a limit on the number of threads, and this is also influenced by stack size and amount of virtual memory available in the server. In addition, there is a context switching cost associated with multiple threads, as when a CPU switches from one thread to another, it needs to save the local data, program pointer etc, of the current thread, and load the program pointer and data for the next thread. Overall using operating system threads incurs cost of context switching, and also some resource costs in the operating system to manage the threads.

So, multi-threading, while suitable for certain scenarios, is not the perfect solution for all situations that require concurrent processing.

Figure 10.4. Async in HTTP request processing



- 1** Users send HTTP requests to the web server simultaneously
- 2** A new async task is spawned to process each incoming request
- 3** The async tasks are managed by the *async run-time* (e.g. Tokio)
- 4** The *async runtime* schedules the async tasks for execution on the processor/core. When *task 1* waits for database operation to complete, the *async run-time* schedules the next task for execution. When the blocking operation in *task 1* completes, the *async runtime* is notified, which then re-schedules *task 1* on the processor for completion.

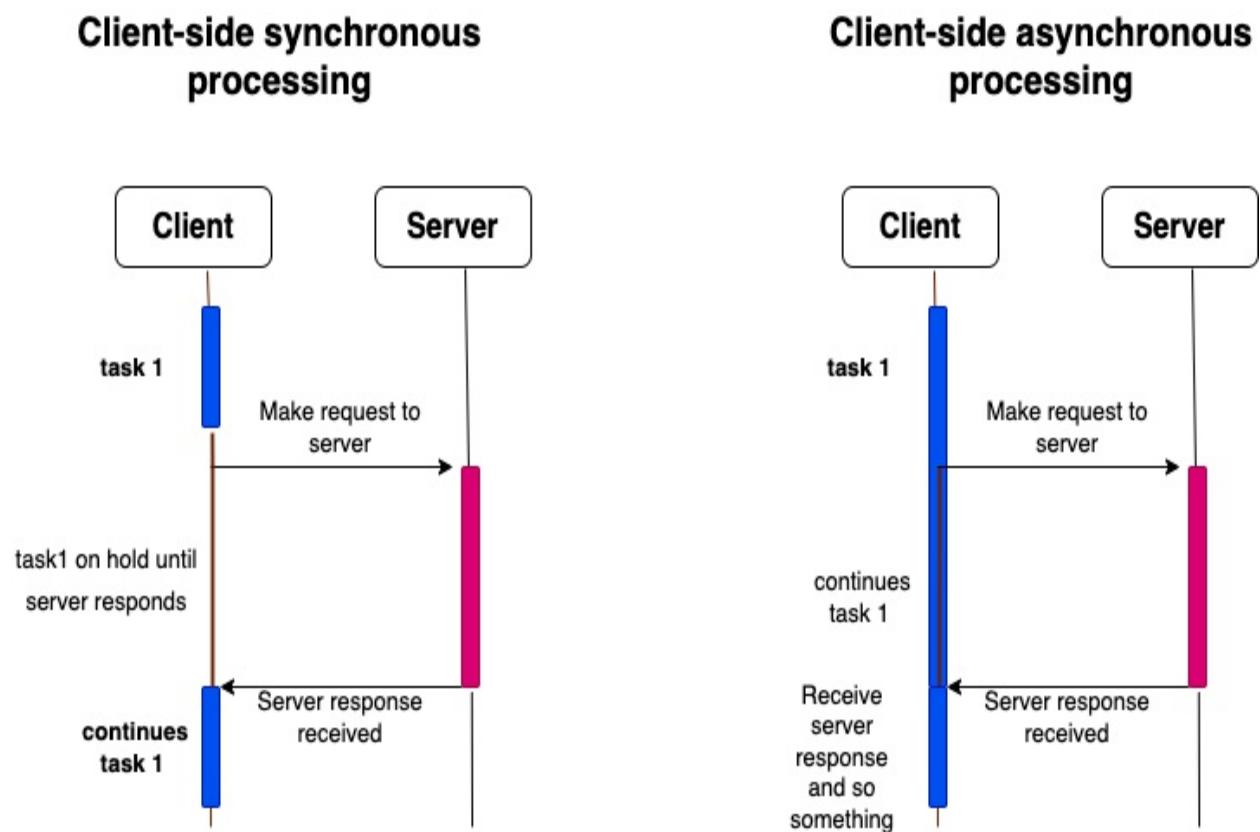
The second approach to concurrent programming (which is also getting popular over the last several years in mainstream programming languages) is **Asynchronous programming** (or **Async** in short). This is illustrated in *figure 10.4* for the scenario of web request processing.

In web applications, *async* programming can be used on both the *client-side* and *server-side*.

Figure 10.4 shows how async processing can be used by an API server/web service to handle multiple incoming requests concurrently, on the *server-side*. Here, as each HTTP request is received by the async web server, it spawns a new async task to handle it. The scheduling of various async tasks on the available CPU(s) is handled by the async runtime.

Figure 10.5 shows how async looks on the *client-side*. Let's consider the example of a javascript application running within a browser trying to upload a file to the server. Without concurrency, the screens would freeze for the user until the file is uploaded and response is received from the server, and the user wouldn't be able to do anything else during this period. With *async* on the client-side, the browser-based UI can continue to process user inputs, while waiting for the server to respond to the previous request.

Figure 10.5. Client-side async processing



We've until now seen the differences between *synchronous*, *multi-threaded* and *async programming* using several examples. Let's next learn how to

implement these different techniques in code.

10.2 Writing concurrent programs

In this section, we'll see how to write *synchronous*, *multi-threaded* and *async* programs in Rust.

We're going to dive into some beginner code straight away showing *synchronous* processing.

Start a new project with

```
cargo new --bin async-hello  
cd async-hello
```

Add the following code to *src/main.rs*:

```
fn main() {  
    println!("Hello before reading file!");  
    let file_contents = read_from_file();  
    println!("{}:", file_contents);  
    println!("Hello after reading file!");  
}  
  
fn read_from_file() -> String {  
    String::from("Hello, there")  
}
```

This is a simple Rust program. It has a function *read_from_file()* that simulates reading a file and returning the contents. This function is invoked from the *main()* function. Note that the call from the *main()* function to *read_from_file()* function is synchronous, i.e., the *main()* function waits for the called function to finish execution and return, before continuing with the rest of the *main()* program.

Run the program with:

```
cargo run
```

You should see the following printed out to your terminal:

```
Hello before reading file!
"Hello, there"
Hello after reading file!
```

There's nothing special with this program. Now let's simulate some delay in reading the file by adding a timer. Modify *src/main.rs* to look like this:

```
use std::thread::sleep;                      #1
use std::time::Duration;                     #2

fn main() {
    println!("Hello before reading file!");
    let file_contents = read_from_file();
    println!("{}:?", file_contents);
    println!("Hello after reading file!");
}

// function that simulates reading from a file
fn read_from_file() -> String {
    sleep(Duration::new(2, 0));                  #3
    String::from("Hello, there")
}
```

Note that the *main()* function still only synchronously calls the *read_from_file()* function, i.e. it waits until the called function is complete (including the delay introduced) before printing out the file contents.

Run the program with:

```
cargo run
```

You can now see the final print statement on your terminal after the specified timer delay period.

Let's add another computation to the mix. Modify the program in *src/main.rs* as shown:

```
use std::thread::sleep;
use std::time::Duration;

fn main() {
    println!("Hello before reading file!");
    let file1_contents = read_from_file1();        #1
    println!("{}:?", file1_contents);
```

```

    println!("Hello after reading file1!");
    let file2_contents = read_from_file2();           #2
    println!("{:?}", file2_contents);
    println!("Hello after reading file2!");
}

// function that simulates reading from a file
fn read_from_file1() -> String {
    sleep(Duration::new(4, 0));
    String::from("Hello, there from file 1")
}

// function that simulates reading from a file
fn read_from_file2() -> String {
    sleep(Duration::new(2, 0));
    String::from("Hello, there from file 2")
}

```

Run the program again, and you'll see that there is a 4-second delay in the execution of the first function and a 2-second delay for the second function, amounting to a total delay of 6 seconds. Can we not do better?

Since the two files are distinct, why can we not read the two files at the same time? Can we use a concurrent programming technique here? Sure, we can use the native operating system threads to achieve this. Modify the code in *src/main.rs* as shown:

```

use std::thread;
use std::thread::sleep;
use std::time::Duration;

fn main() {
    println!("Hello before reading file!");
    let handle1 = thread::spawn(|| {                      #1
        let file1_contents = read_from_file1();
        println!("{:?}", file1_contents);
    });
    let handle2 = thread::spawn(|| {                      #2
        let file2_contents = read_from_file2();
        println!("{:?}", file2_contents);
    });
    handle1.join().unwrap();                            #3
    handle2.join().unwrap();                            #4
}

```

```

// function that simulates reading from a file
fn read_from_file1() -> String {
    sleep(Duration::new(4, 0));
    String::from("Hello, there from file 1")
}

// function that simulates reading from a file
fn read_from_file2() -> String {
    sleep(Duration::new(2, 0));
    String::from("Hello, there from file 2")
}

```

Run the program again. This time you'll see that it does not take 6 seconds for the two functions to complete execution, but much less because both the files are being read concurrently in two separate operating-system threads of execution.

We've just seen concurrency in action using multi-threading.

What if there was another way to process the two files concurrently on a single thread? Let's explore this further using *asynchronous programming* techniques.

For writing basic *multi-threaded* programs, the Rust standard library itself contains the needed primitives (even though external libraries such as *rayon* are available that have additional features). However, for writing and executing *async* programs, only bare essentials are provided by the Rust standard library which is not adequate, and this necessitates the use of external *async* libraries. In this chapter we will make use of *tokio* *async* runtime to illustrate how *asynchronous programs* can be written in Rust.

Add the following to cargo.toml

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

Modify the *src/main.rs* file as shown:

```
use std::thread::sleep;
use std::time::Duration;

#[tokio::main] #1
```

```

async fn main() {                                     #2
    println!("Hello before reading file!");

    let h1 = tokio::spawn(async {                  #3
        let _file1_contents = read_from_file1();
    });

    let h2 = tokio::spawn(async {                  #3
        let _file2_contents = read_from_file2();
    });
    let _ = tokio::join!(h1, h2);                  #4
}

// function that simulates reading from a file
async fn read_from_file1() -> String {           #5
    sleep(Duration::new(4, 0));
    println!("{}:", "Processing file 1");
    String::from("Hello, there from file 1")
}

// function that simulates reading from a file
async fn read_from_file2() -> String {           #5
    sleep(Duration::new(2, 0));
    println!("{}:", "Processing file 2");
    String::from("Hello, there from file 2")
}

```

You'll see many similarities to the previous *multithreaded* example. New *async* tasks are spawned similar to spawning new threads. The *join!* macro waits for all the *async* tasks to complete before completing execution of the *main()* function.

However you'll also notice a few key differences. All the functions including *main()* have been prefixed with the *async* keyword. Another key difference is the annotation *#tokio::main*. We'll delve deeper into these concepts shortly, but let's first try to execute the program.

Run the program with *cargo run* and you'll see the following message printed to the terminal:

```
Hello before reading file!
```

The statement is printed from the *main()* function. But you will notice that the print statements from the two functions *read_from_file_1()* and

`read_from_file_2()` are not printed. It means the functions are not even executed.

The reason is that in Rust, *asynchronous* functions are *lazy*, in that they are executed only when activated with the `.await` keyword.

Let's try this one more time and add the `await` keyword in the call to the two functions. Change the code in `src/main.rs` as shown:

```
use std::thread::sleep;
use std::time::Duration;

#[tokio::main]
async fn main() {
    println!("Hello before reading file!");

    let h1 = tokio::spawn(async {
        let file1_contents = read_from_file1().await;      #1
        println!("{}:", file1_contents);
    });

    let h2 = tokio::spawn(async {
        let file2_contents = read_from_file2().await;      #1
        println!("{}:", file2_contents);
    });
    let _ = tokio::join!(h1, h2);
}

// function that simulates reading from a file
async fn read_from_file1() -> String {
    sleep(Duration::new(4, 0));
    println!("{}:", "Processing file 1");
    String::from("Hello, there from file 1")
}

// function that simulates reading from a file
async fn read_from_file2() -> String {
    sleep(Duration::new(2, 0));
    println!("{}:", "Processing file 2");
    String::from("Hello, there from file 2")
}
```

Run the program again. You should see the following output on your terminal:

```
Hello before reading file!
"Processing file 2"
"Hello, there from file 2"
"Processing file 1"
"Hello, there from file 1"
```

Let's see what just happened. Both the functions called from `main()` are spawned as separate *asynchronous tasks* on the Tokio runtime, which schedules execution of both functions concurrently (analogous to running the two functions on two separate threads). The difference is that these two tasks can both be scheduled either on the current thread or on different threads depending on how we configure the *Tokio* runtime. You'll also notice that the function `read_from_file2()` completes execution before `read_from_file1()`. This is because the sleep time interval for the former is 2 seconds while for the latter it's 4 seconds. So, even though the `read_from_file1()` was spawned earlier to the `read_from_file2()` function in the `main()` function, the `async` runtime executed `read_from_file2()` first because it woke up from the `sleep` interval earlier than `read_from_file1()`.

In this section, we've seen simple examples of how to write *synchronous*, *multi-threaded* and *async* programs in Rust. In the next section, let's go down the *async* Rust rabbit hole.

10.3 Diving deeper into `async` Rust

As seen earlier, asynchronous programming allows us to process multiple tasks at the same time on a single operating system thread. But how is this possible? A CPU can only process one set of instructions at a time, right?

The trick to achieve this is to exploit situations in code execution when the CPU is waiting for some external event or action to complete. Examples could be waiting to read/write a file to disk, waiting for bytes to arrive on a network connection, or waiting for timers to complete (like we saw in the previous example). So, while a piece of code or a function is idle waiting on a disk subsystem or network socket for data, the `async` runtime (such as *Tokio*) schedules other `async` tasks on the processor that are able to continue execution. When the system interrupts arrive from the disk or I/O subsystems, the `async` runtime recognizes this, and schedules the original task

to continue processing.

As a general guideline, programs that are *I/O bound* (i.e. rate of progress of the program depends on the speed of the I/O subsystem) may be good candidates for asynchronous task execution as opposed to *CPU-bound* tasks (i.e. rate of progress of a program is dependent on the speed of the CPU, as in the case of complex number-crunching). Note that this is a broad and general guideline, but as always, there are exceptions.

Since we deal a lot with *network I/O* and *file/database I/O* in web development, *asynchronous programming*, if done right, can speed up overall program execution and improve response times for end users. Imagine a case where your web server has to handle 10,000 or more concurrent connections. Using *multithreading* to spawn a separate OS thread per connection would be prohibitively expensive from a system resource consumption perspective. Actually, early web servers used this model, but then hit the limitations when it came to web-scale systems. This is the reason *Actix web* framework (and many other Rust frameworks) have an *async runtime* built into the framework. As a matter of fact, *Actix web* uses the *Tokio* library underneath, for *asynchronous* task execution (with some modifications/enhancements).

Async/.await keywords represent the core built-in set of primitives in the Rust standard library for *asynchronous* programming. They are just special Rust syntax that make it easier for Rust devs to write *asynchronous* code that looks like *synchronous* code.

However at the core of Rust *async* is a concept called *futures*. *Futures* are single eventual values produced by an asynchronous computation (or function). *Async* functions in Rust return a *future*. *Futures* basically represent deferred computations.

Promises in Javascript

In Javascript, the analogous concept to a Rust future is a promise. When javascript code is executed within a browser, and when a user makes a request to fetch a URL or load an image, it does not block the current thread. The user can continue to interact with the web page. This is achieved by the javascript engine (e.g. V8 in Chrome browser) using asynchronous

processing for network fetch requests. However, note that a Rust future is a lower-level concept than a promise in javascript. A Rust future is something that can be polled for readiness, while a javascript promise has higher semantics (e.g., a promise can be rejected). However, in the context of this discussion, this analogy is useful.

Does this mean our previous program actually used *futures*? Short answer is , yes. Let's rewrite the program to show usage of futures.

```
use std::thread::sleep;
use std::time::Duration;
use std::future::Future;

#[tokio::main]
async fn main() {
    println!("Hello before reading file!");

    let h1 = tokio::spawn(async {
        let file1_contents = read_from_file1().await;
        println!("{}:", file1_contents);
    });

    let h2 = tokio::spawn(async {
        let file2_contents = read_from_file2().await;
        println!("{}:", file2_contents);
    });
    let _ = tokio::join!(h1, h2);
}

// function that simulates reading from a file
fn read_from_file1() -> impl Future<Output=String> { #1
    async { sleep(Duration::new(4, 0)); #2
        println!("{}:", "Processing file 1");
        String::from("Hello, there from file 1")
    }
}

// function that simulates reading from a file
fn read_from_file2() -> impl Future<Output=String> { #1
    async {
        sleep(Duration::new(3, 0));
        println!("{}:", "Processing file 2");
        String::from("Hello, there from file 2")
    }
}
```

Run the program. You should see the same result as earlier.

```
Hello before reading file!
"Processing file 2"
"Hello, there from file 2"
"Processing file 1"
"Hello, there from file 1"
```

The main change we've made to the program is within the two functions - *read_from_file1()* and *read_from_file2()*. The first difference you'll notice is that the return value of the function has changed from *String* to *impl Future<Output=String>*. This is a way of saying that the function returns a *future*, or more specifically, something that implements the *Future* trait.

Async keyword defines an *async* block or function. Specifying this keyword on a function or a code block instructs the compiler to transform the code into something that generates a *future*. This is the reason the following two types of function signatures are analogous:

Function example 1:

```
async fn read_from_file1() -> String {
    sleep(Duration::new(4, 0));
    println!("{}:", "Processing file 1");
    String::from("Hello, there from file 1")
}
```

Function example 2:

```
fn read_from_file1() -> impl Future<Output=String> {
    async { sleep(Duration::new(4, 0));
        println!("{}:", "Processing file 1");
        String::from("Hello, there from file 1")
    }
}
```

Using the *async* keyword in *example 1* is just syntactic sugar for writing code shown in *example 2*.

Let's see what the *Future* trait looks like:

```
pub trait Future {
```

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

A *future* represents an *asynchronous* computation. The *Output* type represents the data type returned when a *future* successfully completes. In our example we are returning a *String* data type from the function, and so we specified the function return value as *impl Future<Output=String>*.

The *poll* method is critical to the functioning of the *asynchronous* program. This method is called by the *Async runtime* to check if the asynchronous computation has completed. The *poll* function returns a data type which is of *enum* type, which can have one of two possible values:

```
Poll::Pending      #1
Poll::Ready(val)   #2
```

The question then would be, who calls the *poll* function? Rust futures are lazy, as we saw earlier where the following statement did not execute:

```
let h1 = tokio::spawn(async {
    let _file1_contents = read_from_file1();
});
```

Rust futures need someone to constantly follow-up with them for completion. Like a project manager who micromanages!

This role is performed by an *async executor*, which is part of the *async Runtime*. The future executors take a set of futures and take them to completion by calling *poll* on them. In our case, the Tokio library has a future executor that performs this function. This is the reason we annotate the function with the *async* keyword:

```
async fn read_from_file1() -> String {
    sleep(Duration::new(4, 0));
    println!("{}:{}", "Processing file 1");
    String::from("Hello, there from file 1")
}
```

or write *async* code block within a function to achieve the same effect like this:

```

fn read_from_file1() -> String {
    async {
        sleep(Duration::new(4, 0));
        println!("{}: {}", "{:?}", "Processing file 1");
        String::from("Hello, there from file 1")
    }
}

```

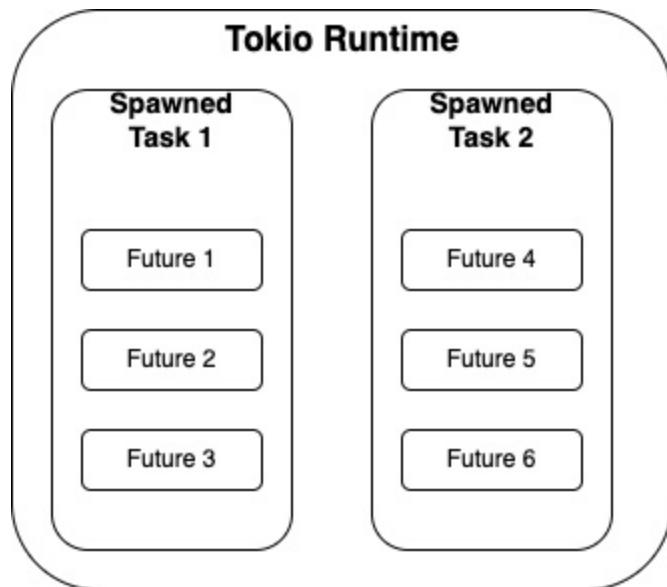
The `async` keyword in front of a function or a code block tells the *Tokio executor* that a future is returned which needs to be driven to completion. But how does the *Tokio executor* know when the `async` function is ready to yield a value? Does it keep polling the `async` function repeatedly? To understand how the *Tokio executor* does this, let's take a closer look at *futures* in the next section.

10.4 Understanding futures

To understand futures better, let's use the concrete example of *Tokio* `async` library.

Figure 10.6 shows the relationship between Tokio runtime, spawned task and a future.

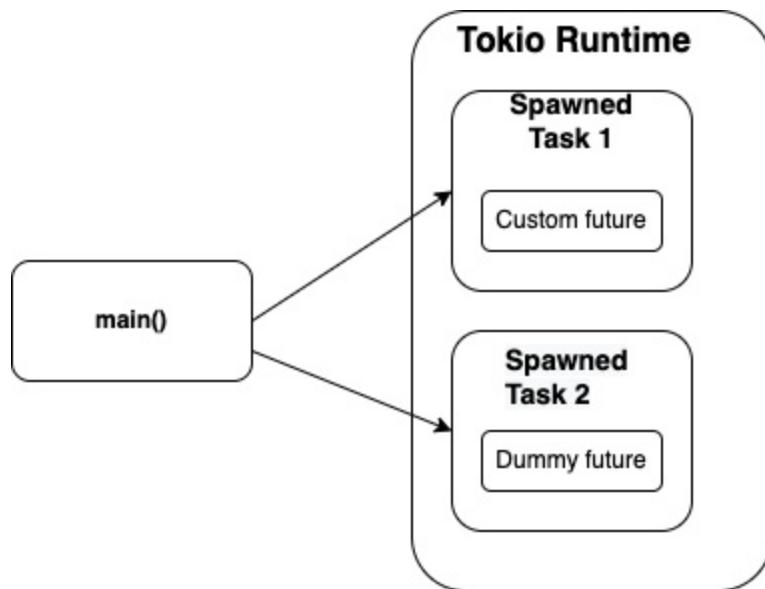
Figure 10.6. Tokio executor



The tokio runtime is the component that manages the *async tasks* and schedules them on the processor for execution. There can be several *async tasks* spawned in a given program. Each *async task* may contain one or more *futures* that return a *Poll::Ready* when the future is ready to be executed or a *Poll::Pending* when it is waiting for an external event (eg a network packet to arrive or a database to return a value).

In the previous section we wrote a *main()* program that spawned two *async tasks* which simulated (dummy) *futures*. In this section and the next, we'll write code that will help us better understand how *futures* work. In this section, we'll see the structure of a *future*, and in the next section, we'll write a custom *async timer* as a *future*. The program will look like what's shown in figure 10.7.

Figure 10.7. Custom Future



What is the purpose of writing a custom future? It's the best way to understand how a future works. So, let's write one.

Modify *src/main.rs* to look as shown:

```
use std::future::Future;
use std::pin::Pin; #1
use std::task::{Context, Poll}; #2
use std::thread::sleep;
```

```

use std::time::Duration;

struct ReadFileFuture {} #3

impl Future for ReadFileFuture { #4
    type Output = String; #5

    fn poll(self: Pin<&mut Self>, _cx: &mut Context<'_>) ->
        Poll<Self::Output> { #6
            println!("Tokio! Stop polling me");
            Poll::Pending
        }
}

#[tokio::main]
async fn main() {
    println!("Hello before reading file!");

    let h1 = tokio::spawn(async {
        let future1 = ReadFileFuture {};
        future1.await
    }); #7

    let h2 = tokio::spawn(async {
        let file2_contents = read_from_file2().await;
        println!("{}:", file2_contents);
    });
    let _ = tokio::join!(h1, h2);
}

// function that simulates reading from a file
fn read_from_file2() -> impl Future<Output = String> {
    async {
        sleep(Duration::new(2, 0));
        println!("{}:", "Processing file 2");
        String::from("Hello, there from file 2")
    }
}

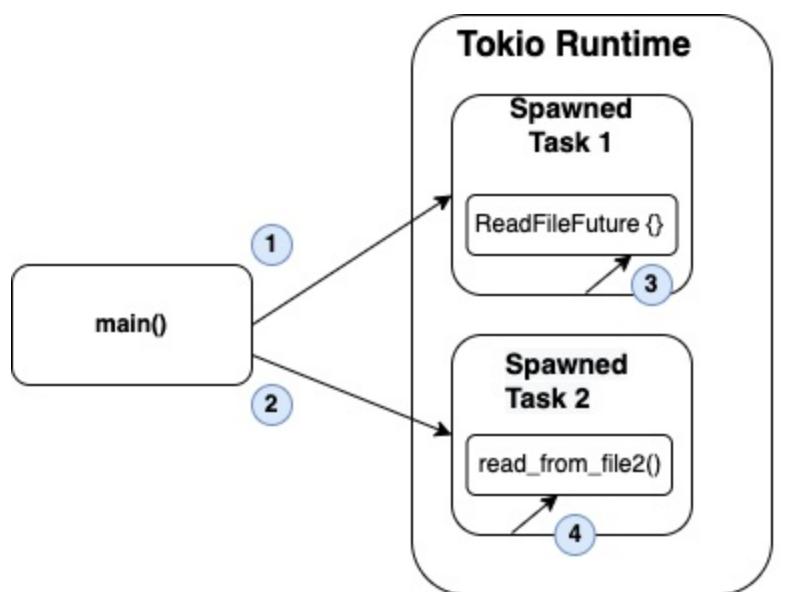
```

Referring back to annotation <1>, we've introduced a new concept of a *Pin*. The reason is that, futures have to be polled repeatedly by the async runtime, so pinning futures to a particular spot in memory is necessary for safe functioning of the code within the async block. This is an advanced concept, so for now it would suffice to treat it as a technical requirement in Rust to write *futures*, even if you do not understand it fully.

Referring to annotation <6>, the `poll()` function is called by *Tokio executor* in its attempt to resolve the future into a final value (of type `String`, in our example). If the future value is not available, the current task is registered with the Waker component, so that when the value from the future becomes available, the Waker component can inform the Tokio runtime to call the `poll()` function again on the future. The `poll()` function returns one of two values: `Poll::Pending` if the future is not ready yet, or `Poll::Ready(future_value)` if the `future_value` is available from the function.

Figure 10.8 illustrates the sequence of steps in program execution:

Figure 10.8. Future- Step 1



- ① main() program spawns first async task
- ② main() program spawns second async task
- ③ First spawned task invokes the custom future. It gets back `Poll::Pending` from the future.
- ④ Second spawned task invokes the async function `read_from_file2()`. The function returns after timer wait of 2 seconds.

future vs Future

If you are confused between a **future** and a **Future**, recall that a **future** is an asynchronous computation that can return a value at a future point of time. It returns a **Future** type (or something that implements the **Future** trait). But to return a value, the **future** has to be polled by the async runtime executor.

Notice the changes we've made to the *main()* function, compared to the code in the previous section. The main (pun intended) change is that we've replaced the call to the async function *read_from_file1()* that returns a future of type *impl Future<Output=String>* with a custom implementation that returns a *future* with the same return type *impl Future<Output=String>*,

Run the program and you should see the following output on your terminal:

```
Hello before reading file!
Tokio! Stop polling me
"Processing file 2"
"Hello, there from file 2"
```

You'll also notice that the program does not terminate and continues to hang as though it's waiting for something.

Referring back to Figure 10.8, let's understand what just happened here. The *main()* function calls two pieces of asynchronous computations (code that returns a *Future*): *ReadFileFuture {}* and *read_from_file2()*. It spawns each of these as asynchronous tasks on the Tokio runtime. The Tokio executor (part of the Tokio runtime) first polls the first future, which returns *Poll::Pending*. It then polls the second future which yields a value of *Poll::Ready* after the sleep timer expires, and so the corresponding statements are printed to the terminal. Then Tokio runtime continues to wait for the first future to be ready to be scheduled for execution. But this will never happen as we are unconditionally returning *Poll::Pending* from the *poll* function). Also note that once a future has finished, the tokio runtime will not call it again. That's why the second function is executed only once.

How does the tokio executor know when to poll the first future again? Does it keep polling repeatedly? The answer is no, as otherwise we would have seen the print statement within the *poll* function several times on the terminal, but we saw that the *poll* function was executed only once.

The way Tokio (and Rust async design) handles it is using a *Waker* component. When a task that's polled by the async executor is not ready to yield a value, the task is registered with a *Waker*, and a handle to the *Waker* is stored in the *Context* object associated with the task. The *Waker* has a *wake()* method that can be used to tell the async executor that the associated task should be awoken. When the *wake()* method is called, the *Tokio executor* is informed that it's time to poll the async task again by invoking the *poll()* function on the task.

Let's see this in action.

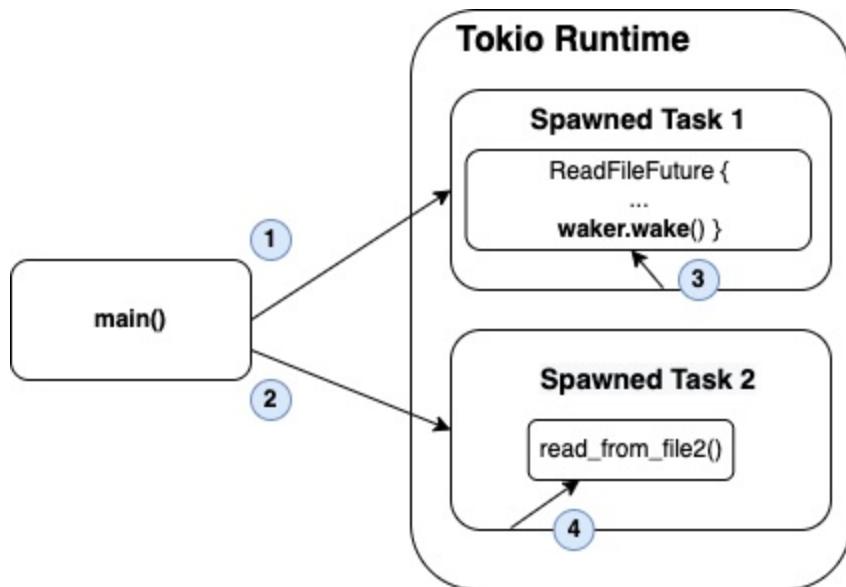
Modify the *poll()* function in *src/main.rs* code as shown:

```
impl Future for ReadFileFuture {
    type Output = String;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
        Poll<Self::Output> { #1
        println!("Tokio! Stop polling me");
        cx.waker().wake_by_ref(); #2
        Poll::Pending
    }
}
```

Figure 10.9 describes this flow.

Figure 10.9. Future- Step 2



- 1 main() program spawns first async task
 - 2 main() program spawns second async task
 - 3 First spawned task invokes the custom future. It invokes **waker,wake()** function which in turn tells the async runtime(tokio) to poll the future again. This loop repeats.
 - 4 Second spawned task invokes the async function `read_from_file2()`. The function returns after timer wait of 2 seconds.

Run the program again, and you should see the `poll()` function being invoked continually. This is because in the `poll` function we are calling the `wake_by_ref()` function on the `Waker` instance, which in turn tells the `async` executor to poll the function again, and the cycle repeats. `wake_by_ref()` function wakes up the task associated with the `Waker`.

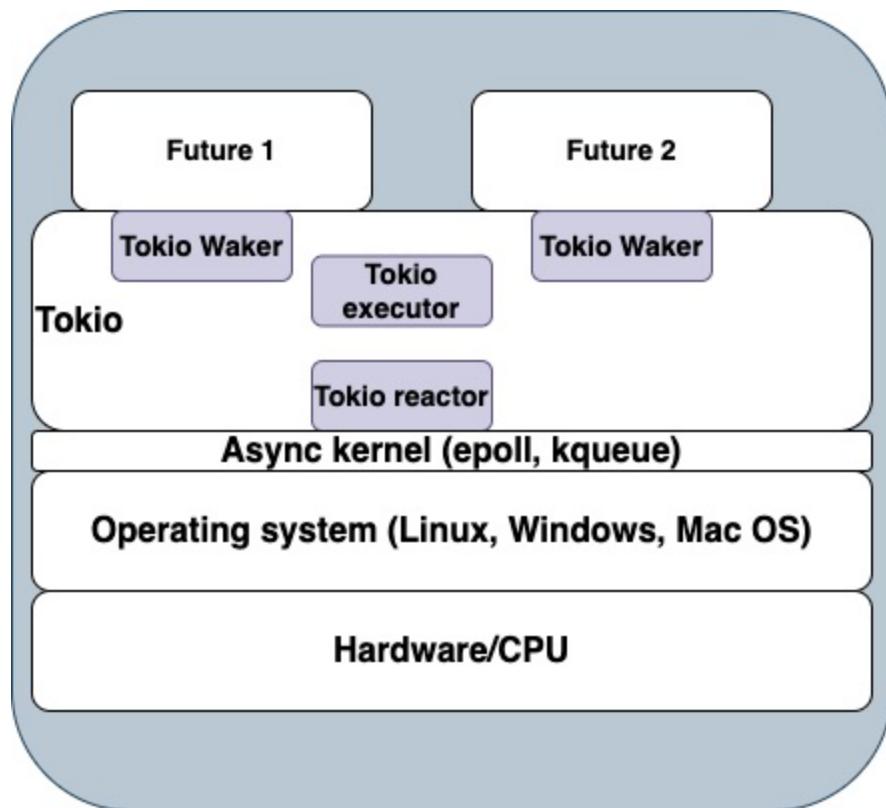
When you run the program, you should see the print statements continually being printed to the terminal as shown, until the program is terminated:

Tokio! Stop polling me

...

Now you may wonder, what is the Waker component? And how does it fit into the Tokio ecosystem? Figure 10.10 shows the various components of Tokio in context of the underlying hardware and operating system.

Figure 10.10. Tokio components



The *Tokio runtime* needs to understand operating system (kernel) methods such as *epoll* to start I/O operations such as reading from a network or writing to a file.

The *Tokio runtime* registers the async handler to be called when an event happens as part of the I/O operation. The component of the Tokio runtime that listens to these events from the kernel and communicates to the rest of the Tokio runtime is the *reactor*.

Tokio executor is the component that takes a future and drives it to completion by calling the *poll()* function of the future, whenever the *future*

can make progress.

How do the *futures* indicate to the *executor* that they are ready to make progress? They call the `wake()` function of the *Waker* component. The *Waker* component informs the *executor*, which then places the *future* back on the queue and invokes the `poll()` function again, until the *future* has completed.

Here is a simplified flow of activities that show how the various Tokio components work together, using an example of reading from a file:

1. Main function of a program spawns *async task 1* on the *Tokio* runtime.
2. *Async task 1* has a *future* that reads data from a large file.
3. The request to read from the file is handed over to the kernel's file subsystem.
4. In the meantime, *async task 2* is scheduled for processing by the *Tokio* runtime.
5. When the file operation associated with *async task 1* is complete, the file subsystem triggers an operating system interrupt, which is translated into an event that is recognized by the *Tokio reactor*.
6. The *Tokio reactor* informs *async task 1* that the data from the file operation is ready.
7. *Async task 1* informs the *Waker* component registered with it, that it is ready to yield a value.
8. The *Waker* component informs the *Tokio executor* to call the `poll()` function associated with *async task 1*.
9. *Tokio executor* schedules *async task 1* for processing, and invokes the `poll()` function.
10. *Async task 1* yields a value.

In summary, the *future*, which performs an I/O operation in an *async* fashion, is informed by the *Tokio reactor* about an I/O event. On receipt of the I/O event, the *future* becomes ready to make progress and invokes the *Tokio Waker* component. The *Waker* component then tells the *Tokio executor* that the future is ready to make progress, which triggers the *Tokio executor* to schedule the future for execution and invoke the `poll()` function on the *future*.

With this background, let's continue with our coding exercise.

Let's now modify the previous program to return a valid value from the `poll()` function, and see what happens. Modify the `poll()` function in `src/main.rs` as shown and rerun the program:

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::thread::sleep;
use std::time::Duration;

struct ReadFileFuture {}

impl Future for ReadFileFuture {
    type Output = String;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
        Poll<Self::Output> {
        println!("Tokio! Stop polling me");
        cx.waker().wake_by_ref();
        Poll::Ready(String::from("Hello, there from file 1")) # 
    }
}

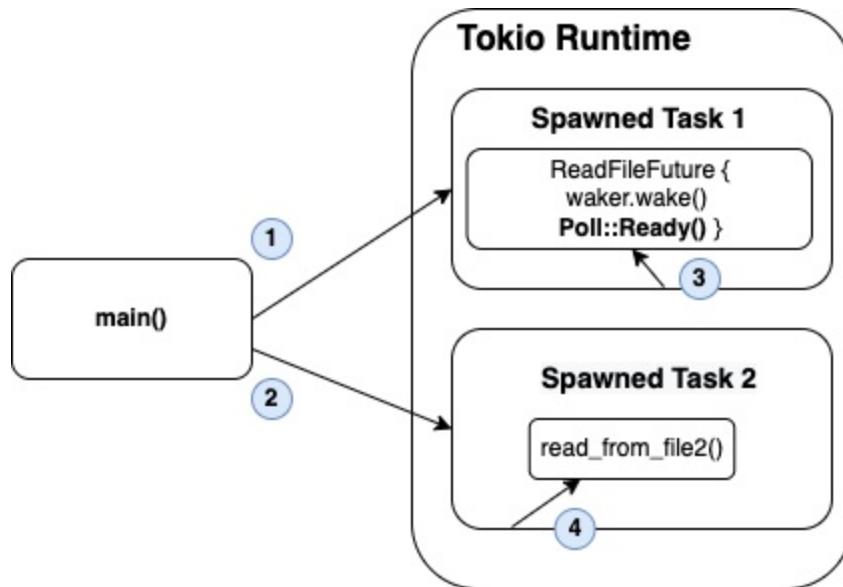
#[tokio::main]
async fn main() {
    println!("Hello before reading file!");

    let h1 = tokio::spawn(async {
        let future1 = ReadFileFuture {};
        println!("{}:", future1.await);
    });

    let h2 = tokio::spawn(async {
        let file2_contents = read_from_file2().await;
        println!("{}:", file2_contents);
    });
    let _ = tokio::join!(h1, h2);
}

// function that simulates reading from a file
fn read_from_file2() -> impl Future<Output = String> {
    async {
        sleep(Duration::new(2, 0));
        String::from("Hello, there from file 2")
    }
}
```

Figure 10.11. Future- Step 3



- 1 main() program spawns first async task
- 2 main() program spawns second async task
- 3 First spawned task invokes the custom future. It returns **Poll::Ready()**
- 4 Second spawned task invokes the async function `read_from_file2()`. The function returns after timer wait of 2 seconds.

You should now see the following output on your terminal:

```
Hello before reading file!
Tokio! Stop polling me
"Hello, there from file 1"
"Hello, there from file 2"
```

The program now does not hang as it completes successfully after executing the two async tasks to completion.

In the next section, we'll take this program one step further and enhance the *future* to implement an *asynchronous timer* functionality. When the time elapses, the *Waker* informs the Tokio runtime that the task associated with it is ready to be polled again. When the Tokio runtime polls the function the

second time, it receives a value from the function.

This should help us understand even better how futures work.

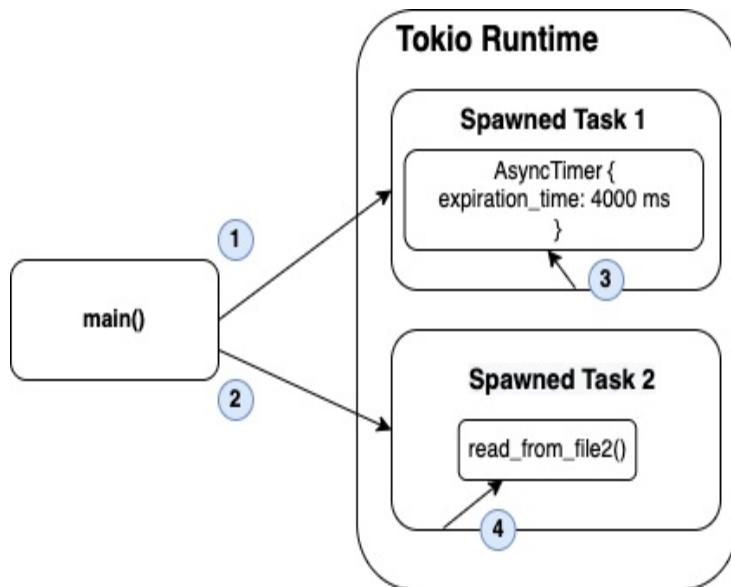
10.5 Implementing a custom future

Let's create a new *future* representing an async timer that performs the following:

1. Accepts an expiration time
2. Whenever it is polled by the runtime executor, it will do the following checks:
 - If the *current time* is \geq *expiration time*, then return *Poll::Ready* with a *String* value
 - If the *current time* $<$ *expiration time*, it will go to sleep until the expiration time, and then trigger the *wake()* call on the *Waker*, which then will inform the async runtime executor to schedule and execute the task again.

Figure 10.12 describes the logic of the custom future in this scenario:

Figure 10.12. Future- Step 4



- 1 `main()` program spawns first async task
- 2 `main()` program spawns second async task
- 3 First spawned task invokes the custom future `AsyncTimer {}`, which takes a parameter `expiration_time`. Everytime the future is polled by Tokio runtime, the expiration time is checked against the current time. If expiration time has not elapsed, `Poll::Pending` is returned. If the expiration time has elapsed, the future returns `Poll::Ready()`, which then tells Tokio runtime to schedule the async task for completion of the remaining execution.
- 4 Second spawned task invokes the async function `read_from_file2()`. The function returns after timer wait of 2 seconds.

Modify `src/main.rs` to look as shown:

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::thread::sleep;
use std::time::{Duration, Instant};

struct AsyncTimer {                                         #1
    expiration_time: Instant,
}

impl Future for AsyncTimer {                                #2
    type Output = String;                                 #3
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) ->
```

```

        Poll<Self::Output> { #4
            if Instant::now() >= self.expiration_time {                      #
                println!("Hello, it's time for Future 1");
                Poll::Ready(String::from("Future 1 has completed"))
            } else {
                println!("Hello, it's not yet time for Future 1. Going
                let waker = cx.waker().clone();
                let expiration_time = self.expiration_time;
                std::thread::spawn(move || {                                     #6
                    let current_time = Instant::now();
                    if current_time < expiration_time {
                        std::thread::sleep(expiration_time - current_time);
                    }
                    waker.wake();
                });
                Poll::Pending
            }
        }
    }

#[tokio::main]
async fn main() {
    let h1 = tokio::spawn(async {
        let future1 = AsyncTimer {                                #
            expiration_time: Instant::now() + Duration::from_millis(1000);
        };
        println!("{}: {:?}", h1, future1.await);
    });

    let h2 = tokio::spawn(async {
        let file2_contents = read_from_file2().await;
        println!("{}: {:?}", h2, file2_contents);
    });
    let _ = tokio::join!(h1, h2);
}

// function that simulates reading from a file
fn read_from_file2() -> impl Future<Output = String> {
    async {
        sleep(Duration::new(2, 0));
        String::from("Future 2 has completed")
    }
}

```

Note that we've implemented a custom future and invoked it within the main function. We've also retained the call to the second future `read_from_file2()` from the main function, that we had implemented earlier. Note that both

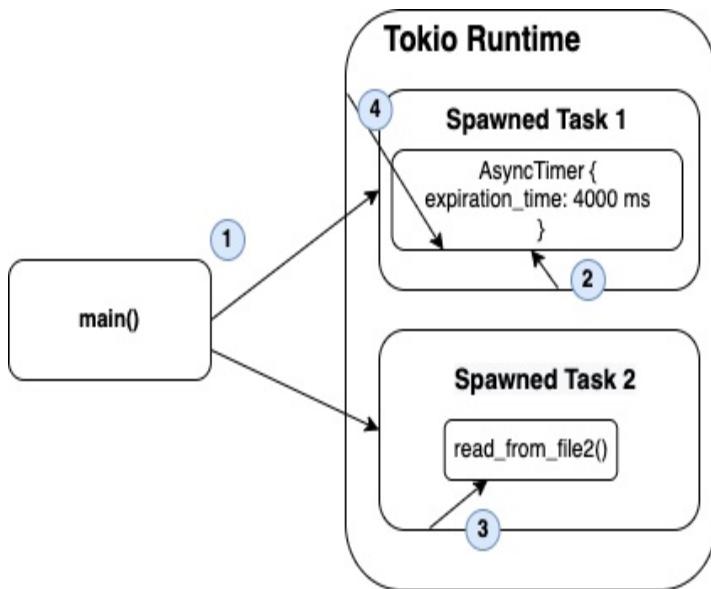
futures eventually implement a timer, but the first future is a fully async way to implement the timer functionality, while the second future simulates an async timer (but uses a synchronous call to the `std::thread::sleep()` internally).

Run the program and you should see the following output on your terminal:

```
Hello, it's not yet time for Future 1. Going to sleep
"Future 2 has completed"
Hello, it's time for Future 1
"Future 1 has completed"
```

Let's analyze what just happened here. Figure 10.13 illustrates the sequence of events.

Figure 10.13. Future- Step 5



- 1 **main()** program spawns first async task, which contains call to the `future: AsyncTimer {}`
- 2 Tokio calls `poll()` function on the future. Since the expiration time is not reached, the future returns `Poll::Pending`. This tells Tokio runtime that future 1 is not ready, and it can schedule other async tasks for execution.
- 3 Tokio schedules the second async task for execution. This task pauses the thread for 2 seconds (sleep timer), and returns control back to Tokio runtime.
- 4 After 4 seconds, the first future is ready to yield a value. the `Waker` tells Tokio runtime that the future can now be polled. Tokio schedules the first async task again for execution.

1. In the `main()` function, the first *async computation* to be scheduled on the async runtime is the call to the future `AsyncTimer`, which is our custom future implementation. Let's call this `future1`.
2. The async executor calls the `poll()` function on `future1`. Since the expiration time has not yet been reached, first the statement "Hello, it's not yet time for Future 1. Going to sleep" is printed to the terminal. The `poll()` function then spawns a new thread and initiates a thread sleep. The `poll` function then returns `Poll::Pending`, which indicates to the executor that other tasks can be scheduled for execution as this async function is not yet ready to yield a value.
3. The async runtime in the meanwhile schedules the task `read_from_file2()` for execution. This function pauses the current thread for 2 seconds and then returns `Poll::Ready` with a *String* value. The print

statement "Future 2 has completed" from this future is printed to the terminal.

4. In the meantime, the first future becomes ready to yield a value. It calls the `wake()` function on the *Waker* associated with this async task, which in turn informs the async executor that *future1* is ready to be scheduled for execution again as it is now ready to yield a value. The executor calls the `poll()` function on *future1*, which now returns `Poll::Ready` with a string value. Note that the following two print statements are printed to the terminal: "Hello, it's time for Future 1" and "Future 1 has completed".

I hope the exercises in this chapter gave you a better understanding of how async functions work and how they are implemented in Rust. In many cases, you may not even implement your own futures, but instead use the dev-friendly APIs provided by async runtimes such as *Tokio*, or by higher level frameworks such as *Actix web*. But it helps to understand how async and futures work under the hood.

With this, we conclude this section on writing a custom future.

10.6 Summary

- In this chapter, we learnt the differences between *concurrency* and *parallelism*
- We learnt the differences between *multithreaded* and *async* models of concurrency with practical examples. We first wrote a basic program to simulate reading from two files using *synchronous* code. Then we converted the program to use a *multithreaded* model of concurrency and observed the difference in execution performance. We then learnt how to write basic *asynchronous* programs using the *Tokio runtime* library.
- We went deeper into the asynchronous programming concepts in Rust including *async/.await*, *futures* and the *Future* trait. We saw examples of how to use them.
- Lastly we wrote a custom *future* that implements the *Future* trait, and specifies the conditions under which the future is not ready to return a value (i,e returns `Poll::Pending`), and when it is ready to yield a value (i,e, `Poll::Ready`). We then learnt how to use the custom future and

schedule it for execution on the async runtime. We analyzed the detailed sequence of events around how the async runtime schedules and executes our custom future to completion.

With this, we come to an end of this chapter. We'll implement a networking project in async Rust in the next chapter.

See you soon!

11 Building a P2P node with Async Rust

This chapter covers

- Introduction to peer-to-peer networks
- Understanding the core architecture of libp2p networking
- Exchanging ping commands between peer nodes
- Discovering peers in a p2p network

In the previous chapter we covered the basics of async programming in general, and how to write async code with Rust. In this chapter we'll build a few simple examples of p2p applications using a low-level P2P networking library and asynchronous programming using Rust.

But why learn about P2P?

P2P is a networking technology that enables sharing of various computing resources such as CPU, network bandwidth and storage across different computers. P2P is today a very commonly used method for sharing files (such as music, images and other digital media) between users online.

Bittorrent and Gnutella are examples of popular file sharing p2p apps. They do not rely on a central server or an intermediary to connect multiple clients. And most importantly, they make use of users' computers as both clients and servers, thus offloading computations away from a central server. How do p2p networks operate and how are they different?

Let's delve into the foundational concepts behind peer-to-peer networks.

11.1 Introduction to peer-to-peer networks

Traditional distributed systems deployed within the enterprise or the web use the client-server paradigm. A web browser and a web server together serve as

a good example of a *client-server* system where the web browser (the *client*) requests information (e.g., a GET Request), or a computation (e.g., POST/PUT/DELETE requests), on a particular resource hosted on the web server (the *server*). The web server then determines if the client is authorized to receive that information or perform that computation, and then fulfills the request.

Peer to peer networks (P2P) are another type of distributed systems. In P2P, a set of nodes (or peers) interact directly with one another to collectively provide a common service, without having a central coordinator or administrator. Examples of peer to peer systems include file-sharing networks such as IPFS and BitTorrent, and blockchain networks such as Bitcoin and Ethereum. Each node (or peer) in a P2P system can act as both a *client* (requesting information from other nodes) and a *server* (storing/retrieving data and performing necessary computations in response to client requests). While all the nodes in a P2P network need not be identical, one key characteristic that differentiates *client-server* networks from *P2P networks* is the absence of dedicated *servers* that have unique privileges. In open, permissionless P2P networks, any node can decide to offer a full or partial set of services associated with a P2P node.

Compared to *client-server* networks, P2P networks enable a different class of applications to be built over them that are *permissionless*, *fault-tolerant* and *censorship-resistant*.

Permissionless because no server can cut off access to information to a client, as the data and state are replicated across multiple nodes.

Fault-tolerant because there is no single point of failure, such as a central *server*.

Censorship-resistant as in networks such as blockchains.

P2P computing also enables better utilization of resources. Imagine all the network bandwidth, storage, processing power that's available with the clients at the edge of the network that are not utilized in *client-server* computing.

Figure 11.1. Client-server vs peer-to-peer computing

Peer to peer systems

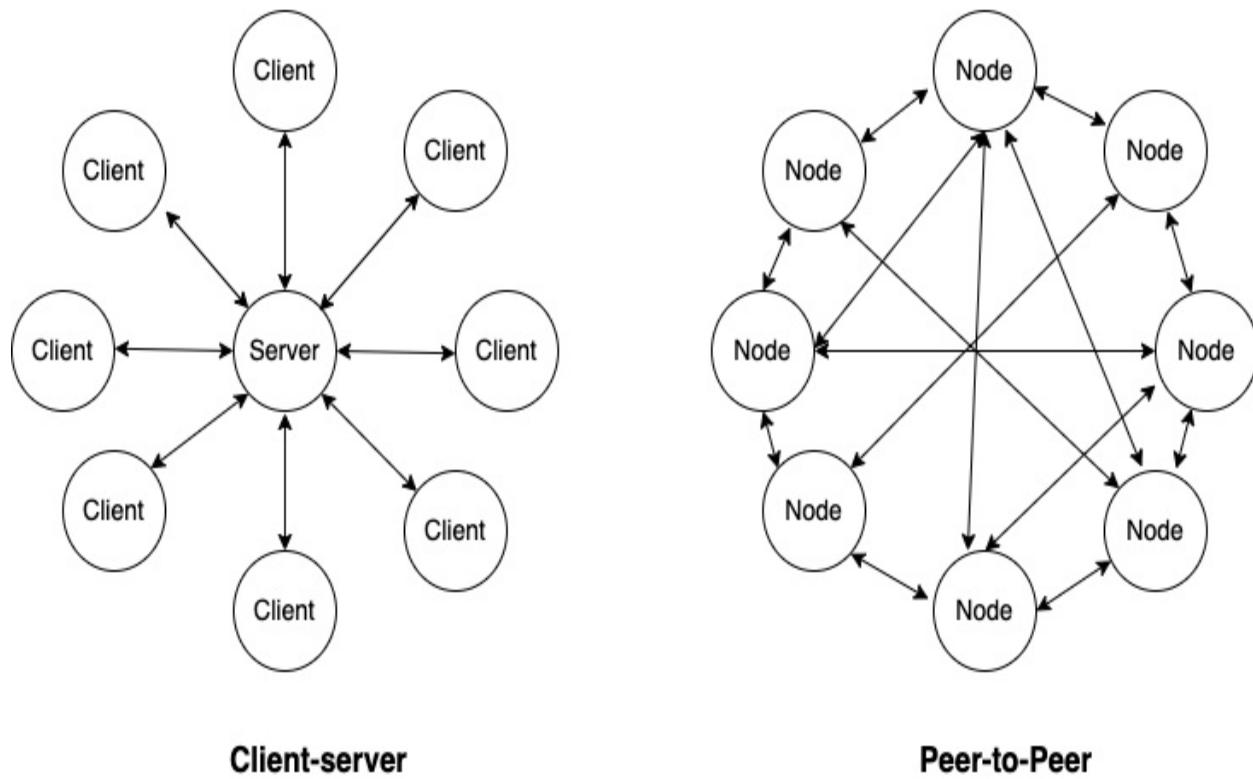


Figure 11.1 illustrates the differences between a *client-server* and a *p2p network*. Note that we will use the terms *node* and *peer* interchangeably in the context of a *p2p network*.

However building *P2P systems* can be more complex than traditional *client-server* systems. Some of the technical requirements associated with building *P2P systems* include:

- *Transport:* Each peer in a P2P network can speak a different protocol, e.g. HTTP(s), TCP, UDP, etc.
- *Identity:* Each peer needs to know the identity of the peer to which it wants to connect and send a message.
- *Security:* Each peer should be able to communicate with other peers in a secure manner without the risk of a third-party intercepting or modifying

messages

- *Peer routing:* Each peer can receive a message from other peers through a variety of routes (like how data packets are distributed in IP protocol), which means that each peer should have the ability to route the message to other peers if the message is not intended for itself.
- *Messaging:* P2P networks should be able to send point-to-point messages or group messages (in a *publish/subscribe* pattern)

Let's take a closer look at each of these requirements:

Transport: The TCP/IP and UDP protocols are ubiquitous and are popular for writing networked applications. But there are other higher-level protocols such as HTTP (layered over TCP) and QUIC (layered over UDP). Each peer in a P2P network should have the ability to initiate a connection to another node, and be able to listen to incoming connections over multiple protocols because of the diversity of peers in the network.

Peer identity: Unlike the web development domain where a server is identified by a unique domain name (such as www.rust-lang.org, which is then resolved to the IP address of the server using a *Domain name service*), nodes in a peer-to-peer network need a unique identity so that the other nodes can reach them. Nodes in a peer-to-peer network use a *public* and *private* key pair (asymmetric public key cryptography) to establish secure communications with other nodes. The identity of a node in a peer-to-peer network is called the *PeerId*, which is a cryptographic hash of the node's public key.

Security: The cryptographic key pair and *PeerId* enable a node to establish secure, authenticated communication channels with its peers. But that's only one aspect of security. Nodes also need to implement frameworks for authorization, which establish rules for what kinds of operations can be performed by which node. There are also network level security threats to be addressed such as *sybil attacks* (where one of the node operators spins up a large number of nodes with distinct identities to gain an advantageous position in the network), or *eclipse attacks* (where a group of malicious nodes collude to target a specific node such that the latter cannot reach any legitimate nodes). More information about differences between *Sybil* and *eclipse* attacks can be found publicly on the internet.

Peer routing: A node in a P2P network first needs to find other peers in order to communicate. This is achieved by maintaining a *peer routing* table, which contains references to other peers in the network. But in a P2P network that has thousands of nodes or more that are changing dynamically (i.e. nodes join and leave the network), it is difficult for any single node to maintain a complete and accurate *routing table* for all nodes in the network. *Peer routing* enables nodes to route messages that are not meant for them, to the destination nodes.

Messaging: Nodes in a P2P network can send messages to specific nodes, but can also participate in *broadcast* messaging protocols. An example is *publish/subscribe* where nodes register interest in a particular topic (*subscribe*) and any node that sends messages on that topic (*publish*) is received by all the nodes that *subscribe* to that topic. This technique is commonly used to transmit the contents of a message to the entire network. Note that *publish/subscribe* is a well known architectural pattern for messaging in a distributed system, between a sender and a receiver.

Stream multiplexing: We've previously seen (in the earlier paragraph on *transport*) how a node in a P2P network can support multiple *transports*. *Stream multiplexing* is a way to send multiple streams of information over a common communication link. In case of P2P, it allows multiple independent 'logical' streams to share a common P2P transport layer. This becomes important when considering the possibility of a node having multiple streams of communications with different peers, or the possibility that there can also be many concurrent connections between two remote nodes. *Stream multiplexing* helps to optimize the overheads of establishing connections between peers. (Note: Multiplexing is common in backend services development where a client can establish an underlying network connection with a server, and then multiplex different streams (each with unique port numbers) over the underlying network connection).

In this section, we have looked at a few foundational concepts that are involved in the design of *peer-to-peer* systems. In the next section, we'll take a closer look at a popular Rust library that is used for P2P networking, as we will be using this library to write some async Rust code in later sections.

11.2 Understanding the core architecture of libp2p networking

Writing your own networking layer for P2P applications is a mammoth task. Also, if someone has already done the hard work, we do not want to reinvent the wheel. So, we will use a low-level p2p networking library called *libp2p* which makes it a lot easier to build P2P applications upon.

To be more specific, *libp2p* is a modular system of protocols, specifications and libraries that enable the development of peer-to-peer applications. *libp2p* supports three programming languages at the time of writing - Go, Javascript and Rust. *libp2p* is used by many popular projects such as *IPFS*, *Filecoin* and *Polkadot*.

Figure 11.2. Components of libp2p

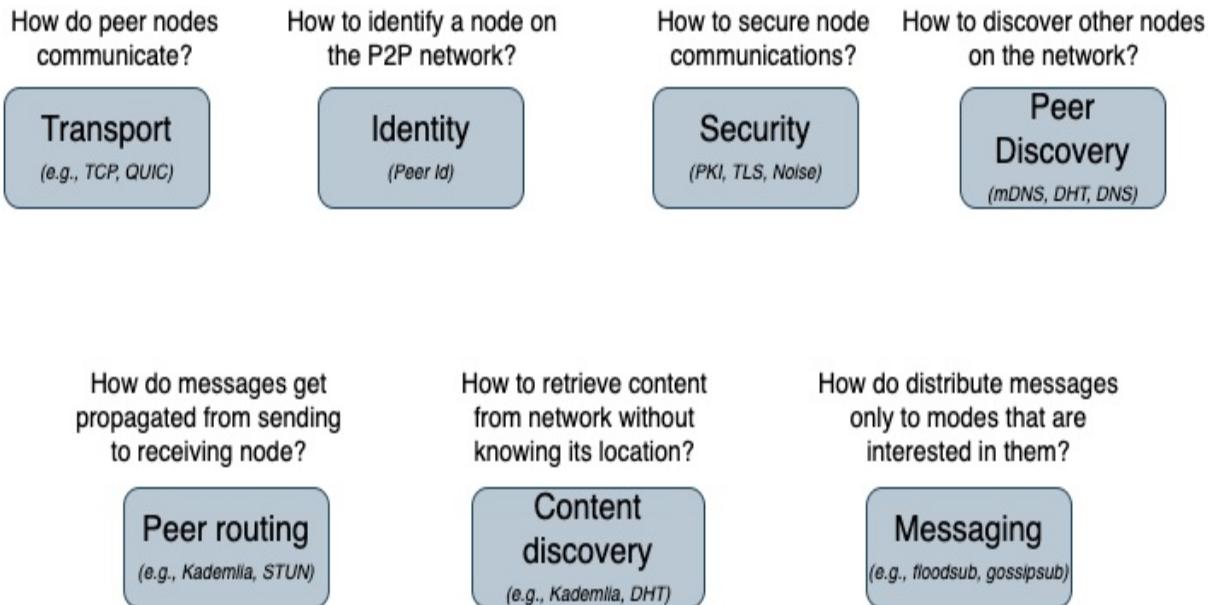


Figure 11.2 highlights the key modules of libp2p that are used to build a robust peer-to-peer network.

Transport: Responsible for the actual transmission and receipt of data from one peer node to another

Identity: libp2p uses public key cryptography (PKI) as the basis of peer node identity. A unique peer id is generated for each node using a cryptographic algorithm.

Security: Nodes sign messages using their private key. Also, the transport connections between nodes can be upgraded into secure encrypted channels so that the remote peers can mutually trust one another, and no third party can

intercept communications between them.

Peer discovery: Enables peers to find and communicate to one another in the libp2p network

Peer routing: Enables communication with a Peer node using the knowledge of other peers.

Content routing: Enables peer nodes to get a piece of content from other peers, without knowing which peer has it

PubSub: Enables sending messages to a group of peers that are interested in a topic.

In this chapter you will learn how to leverage a subset of the features of the libp2p protocol to build p2p applications using Rust.

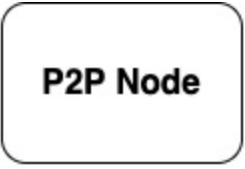
Let's now start by taking a look at a few core primitives of the Rust *libp2p* library, using code examples.

11.2.1 Peer ids and Key Pairs

Let's start with generating *peer ids* and *key pairs* for a P2P node.

Cryptographic *key pairs* are used by p2p nodes to sign messages and *peer ids* represent unique peer identities that are used to uniquely identify nodes on the p2p network.

Figure 11.3. Identity of a p2p node



P2P Node

PeerId: 12D3KooWBu3fmjZgSMLkQ2p1DG35UmEayYBrhsk6WEe1xco1JFbV

A p2p node is identified by a unique PeerId that is cryptographically generated

Start a new project with `cargo new p2p-learn`

In `Cargo.toml`, add the following entry:

```
libp2p = "0.42.2"    #1
tokio = { version = "1.16.1", features = ["full"] }  #1
```

Create a folder *bin* under *src* folder. Create a new file *src/bin/iter1.rs*, and add the following code to it.

```
use libp2p::{identity, PeerId};      #1
#[tokio::main]                      #2
async fn main() {                    #3
    let new_key = identity::Keypair::generate_ed25519();    #4
    let new_peer_id = PeerId::from(new_key.public());        #5
    println!("New peer id: {:?}", new_peer_id);
}
```

What are public and private keys?

Cryptographic identity uses Public Key infrastructure (PKI), which is used widely to provide unique identities for users, devices and applications and to secure end-to-end communications. It works by creating two different cryptographic keys, also known as a *keypair* comprising a private key and a public key, which have a mathematical relationship between them. Keypairs have many wide applications, but in the P2P network, nodes identify and authenticate themselves to each other using keypairs. The public key can be shared with others in a network, but the private key of a node must never be revealed.

A good example of using a keypair is in traditional server access. For example, if you want to connect to a remote server (using ssh) hosted in a data center or a cloud, a keypair can be configured for access instead of using a password. In this example, a user can generate a key pair and configure the public key on the remote server, which grants access to the user. But how does the remote server know which user is the owner of that public key? To enable this, when connecting (over SSH) to the remote server, the user must specify the private key (associated with the public key stored on the server). The private key is never sent to the remote server, but the SSH client (running on the local server) uses the private key of user to authenticate itself to the remote SSH server.

Private and public keys have many other uses, such as for encryption/decryption and digital signatures, but that is out of scope for this chapter.

Run the program with `cargo run --bin iter1`, and you should see something similar to this printed to your terminal:

```
New peer id: PeerId("12D3KooWBu3fmjZgSMLkQ2p1DG35UmEayYBrhsk6WEe1")
```

In *libp2p*, a peer's identity is stable and verifiable for the entire lifetime of a peer. However, *libp2p* makes a distinction between a peer's *identity* and its *location*. As discussed before, the identity of a peer is the *peer id*. The *location* of a peer is the network address at which the peer can be reached. For example, a peer can be reached over TCP, websockets, QUIC or any other protocol. *libp2p* encodes these network addresses in a self-describing format called *multiaddresses* (*multiaddr*). So, in *libp2p*, *multiaddress* represents the location of a peer. We'll look at how to use *multiaddresses* in the next section.

11.2.2 Multiaddresses

When humans share contact information, they use their phone numbers, social media profiles or physical location addresses (in case of receiving delivery of goods). When nodes on a p2p network share their contact information, they send a *multiaddress* containing both the *network address* and their *peer id*.

The *peer id* component of a *multiaddress* for a node is represented like this:

```
/p2p/12D3KooWBu3fmjZgSMLkQ2p1DG35UmEayYBrhsk6WEe1xco1JFbV
```

The string

12D3KooWBu3fmjZgSMLkQ2p1DG35UmEayYBrhsk6WEe1xco1JFbV
represents the *peer id* of the node. Recall that we learnt how to generate the *peer id* for a node in the previous section.

The *network address* component of a *multiaddress* (also known as *transport address*), looks like this:

```
/ip4/192.158.1.23/tcp/1234
```

This says that IPv4 is the transport protocol used, the IP address is 192.158.1.23 and the TCP port on which it listens is 1234.

The complete *multiaddress* of a node is just a combination of the *peer id* and *network address* and looks like this:

```
/ip4/192.158.1.23/tcp/1234/p2p/12D3KooWBu3fmjZgSMLkQ2p1DG35UmEayY
```

Peers exchange this *multiaddress* with other peers, in the format shown here.

The *libp2p* library internally converts this "name-based" address -
/ip4/192.158.1.23 into a regular IP address, using the DNS protocol.

Figure 11.4. Multiaddress of a p2p node

P2P Node

Peer Id:

Identity 12D3KooWByvE1LD4W1oaD2AgeVWAEu9eK4RtD3GuKU1jVEZUvzNm

Multiaddress: /ip4/192.158.1.23/tcp/1234/p2p/12D3KooWByvE1LD4W1oaD2AgeVWAEu9eK4RtD3GuKU1jVEZUvzNm

Identity: A P2P node's identity is used by other peer nodes to identify and address an individual node in the p2p network

Multiaddress: Enables peer nodes to establish connection with other peers on the p2p network. The libp2p protocol converts this multi-address into a regular IP address.

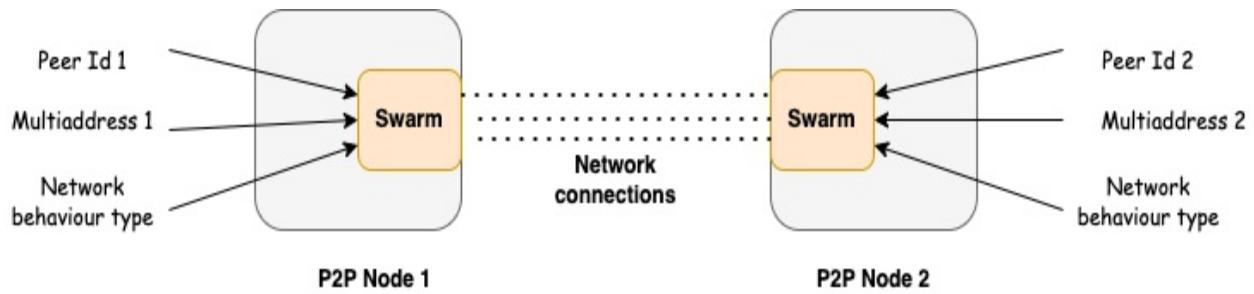
We'll see usage of *multiaddress* in code, in the next section.

11.2.3 Swarm and network behaviour

Swarm is the network manager module within a given P2P node, in libp2p. It maintains all active and pending connections to remote nodes from a given node, and manages the state of all the substreams that have been opened.

The structure and context of Swarm is depicted in figure 11.5, and is explained in detail further in this section.

Figure 11.5. Network management for a p2p node



Function of Swarm

Swarm acts as the network manager for a P2P node , and manages remote connections with other nodes

Structure of Swarm

Swarm object for a P2P node is constructed using a combination of peer Id, multiaddress and network behaviour for that node

Let's now extend the previous example. Create a new file `src/bin/iter2.rs` and add the following code:

```
use libp2p::swarm::{DummyBehaviour, Swarm, SwarmEvent}; #1
use libp2p::futures::StreamExt; #2
use libp2p::{identity, PeerId};
use std::error::Error;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let new_key = identity::Keypair::generate_ed25519();
    let new_peer_id = PeerId::from(new_key.public());
    println!("local peer id is: {:?}", new_peer_id);
    let behaviour = DummyBehaviour::default();
    let transport = libp2p::development_transport(new_key).await?
    let mut swarm = Swarm::new(transport, behaviour, new_peer_id)
    swarm.listen_on("/ip4/0.0.0.0/tcp/0".parse())?;

    loop {
        match swarm.select_next_some().await {
            SwarmEvent::NewListenAddr { address, .. } => {
                println!("Listening on local address {:?}", address)
            }
            _ => {}
        }
    }
}
```

With reference to annotation <1>, it is necessary to construct a swarm

network manager before being able to communicate with other nodes. *Swarm* represents a low-level interface and provides fine-grained control over the *libp2p* network. Swarm is constructed using a combination of transport, network behaviour and peer id for the node. We've previously seen what a *transport* and a *peer_id* is. Let's now look at what is *network behaviour*.

While the *transport* specifies how to send bytes over the network, *network behaviour* specifies what bytes to send and to whom. Examples of network behaviours in *libp2p* include *Ping* (where nodes send and respond to ping messages), *mDNS* which is used to discover other peer nodes on the network and *Kademlia* (used for peer routing and content routing functionality). In our example, to keep it simple to start with, we have used a dummy network behaviour. Multiple network behaviours can be associated with a single running node.

Let's next look at the line of code in annotation <7>:

`swarm.select_next_some().await`. The *await* keyword is used to schedule the asynchronous task to poll the protocols and connections, and when ready, swarm events are received. When there is nothing to process, the task will be idle and the swarm will output *Poll::Pending*. This is another example of async Rust in action.

One thing to note is that the same code runs on all nodes of a *libp2p* network, unlike a client-server model where the *client* and the *server* have different codebases.

Let's run the code as described here.

Create two terminal sessions on your computer. From the first terminal, from the project root directory, run:

```
cargo run --bin iter2
```

You should see an output similar to below printed to your terminal for the first node.

```
local peer id is: PeerId("12D3KooWByvE1LD4w1oaD2AgeVwAEu9eK4RtD3G"
Listening on local address "/ip4/127.0.0.1/tcp/55436"
Listening on local address "/ip4/192.168.1.74/tcp/55436"
```

From the second terminal, from the project root directory, run the following:

```
cargo run --bin iter2
```

You should see terminal output similar to this for the second node:

```
local peer id is: PeerId("12D3KooWQiQZA5zcLzhF86kuRoq9f6yAgiLtGqD"
Listening on local address "/ip4/127.0.0.1/tcp/55501"
Listening on local address "/ip4/192.168.1.74/tcp/55501"
```

Again you can see the local address on which node2 is listening (printed out to the terminal).

If you got so far, it's a good start.

However, there isn't anything interesting happening in this code. We were able to start two nodes and ask them to connect to each other. But we don't know whether the connection has been established correctly, or if the two can communicate. Let's enhance this code in the next section to exchange ping commands between nodes.

11.3 Exchanging ping commands between peer nodes

Create a new file *src/bin/iter3.rs* and add the following code:

```
use libp2p::swarm::{Swarm, SwarmEvent};
use libp2p::futures::StreamExt;
use libp2p::ping::{Ping, PingConfig};
use libp2p::{identity, Multiaddr, PeerId};
use std::error::Error;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let new_key = identity::Keypair::generate_ed25519();
    let new_peer_id = PeerId::from(new_key.public());
    println!("local peer id is: {:?}", new_peer_id);

    let transport = libp2p::development_transport(new_key).await?;
    let behaviour = Ping::new(PingConfig::new().with_keep_alive(true));
    let mut swarm = Swarm::new(transport, behaviour, new_peer_id)
```

```

        swarm.listen_on("/ip4/0.0.0.0/tcp/0".parse()?)?;

    if let Some(remote_peer) = std::env::args().nth(1) {
        let remote_peer_multiaddr: Multiaddr = remote_peer.parse();
        swarm.dial(remote_peer_multiaddr)?;
        println!("Dialed remote peer: {:?}", remote_peer);
    }

    loop {
        match swarm.select_next_some().await {
            SwarmEvent::NewListenAddr { address, .. } => {
                println!("Listening on local address {:?}", address);
            }
            SwarmEvent::Behaviour(event) => println!("Event received: {:?}", event)
            _ => {}
        }
    }
}

```

In annotation <2>, *0.0.0.0* means all IPv4 addresses on the local machine. For example, if a host has two IP addresses, *192.168.1.2* and *10.0.0.1*, and a server running on the host listens on *0.0.0.0*, it will be reachable at both IPs. The *0* port means to choose a random available port.

In annotation <3>, the remote node multiaddress is parsed from the command line parameter. The local node then establishes a connection to the remote node on this multiaddress.

Let's now build and test this p2p example with two nodes.

Create two terminal sessions on your computer. From the first terminal, from the project root directory, run:

```
cargo run --bin iter3
```

Let's call this *node1*.

You should see an output similar to below printed to your terminal for the first node.

```
local peer id is: PeerId("12D3KooWByvE1LD4W1oaD2AgeVWAEu9eK4RtD3G"
Listening on local address "/ip4/127.0.0.1/tcp/55872"
Listening on local address "/ip4/192.168.1.74/tcp/55872"
```

Note that at this point there is no remote node to connect to , so the local node just prints out the listen event along with the multiaddress at which it is listening for new connections. So, the Ping network behaviour, even though it has been configured in the local node, is not active yet. For this, we need to start the second node.

From the second terminal, from the project root directory, run the following. Make sure to specify the multiaddress of the first node in the command line parameter.

```
cargo run --bin iter3 /ip4/127.0.0.1/tcp/55872
```

Let's call this *node2*.

At this point *node2* has started and it will also print out the local address on which it is listening. Since the remote node multiaddress has been specified, *node2* establishes connection with *node1* and then starts to listen to events. On receipt of the incoming connection from *node2*, *node 1* sends the ping message to *node2*, and *node2* responds with a pong message. These messages should start to appear on the terminals of both *node1* and *node2*, and continue in a loop after a time interval (approx every 15 seconds or so). Note also that the P2P node uses async Rust with the Tokio runtime to execute concurrent tasks to process multiple data streams and events coming from remote nodes.

In this section, we have seen how to have two P2P nodes exchange ping messages with each other. In this example, we connected *node2* to *node1* by specifying the multiaddress *node1* is listening on. But in a P2P network, nodes join and leave dynamically. In the next section we'll see how peer nodes can discover each other on a p2p network.

11.4 Discovering peers

Let's code a P2P node this time to automatically detect other nodes on the network, on startup.

```
use libp2p::{
    futures::StreamExt,
    identity,
```

```

mdns::{Mdns, MdnsConfig, MdnsEvent},
swarm::{Swarm, SwarmEvent},
PeerId,
};

use std::error::Error;

#[tokio::main]
async fn main() -> Result<(), Box

```

mDNS is a protocol defined by RFC 6762 (<https://datatracker.ietf.org/doc/html/rfc6762>) which resolves host names to IP addresses. In libp2p, it is used to discover other nodes on the network.

The network behaviour *mDNS* implemented in libp2p will automatically discover other libp2p nodes on the local network.

Let's see this working by building and running the code with:

```
cargo run --bin iter4
```

Let's call this *node1*.

You'll see something similar to this printed to the terminal window of *node1*:

```
Local peer id: PeerId("12D3KooWNgYbVg8ZyJ4ict2N1hdJLKoydB5sTqwiWN  
Listening on local address "/ip4/127.0.0.1/tcp/50960"  
Listening on local address "/ip4/192.168.1.74/tcp/50960"
```

Note that in this example, *node1* is listening on TCP port 50960.

Then from terminal 2, run the program with the same command. Note that unlike before, we are not specifying the multiaddress of node 1.

```
cargo run --bin iter4
```

Let's call this *node2*.

You should be able to see similar messages printed to the terminal of *node2*.

```
Local peer id: PeerId("12D3KooWCVVb2EyxB1WdAcLeMuyaJ7nnfUCq45YNNu  
Listening on local address "/ip4/127.0.0.1/tcp/50967"  
Listening on local address "/ip4/192.168.1.74/tcp/50967"  
discovered 12D3KooWNgYbVg8ZyJ4ict2N1hdJLKoydB5sTqwiWN2ShtC3HwWt /  
discovered 12D3KooWNgYbVg8ZyJ4ict2N1hdJLKoydB5sTqwiWN2ShtC3HwWt /
```

Notice that *node2* was able to discover *node1* listening on port 50960, which is the port on which *node1* is listening on. While *node2* itself is listening to new events and messages on port 50967.

Start a third node (*node3*) from another terminal. You should see the following:

```
cargo run --bin iter4
```

You'll see the following messages on the terminal of *node3*:

```
Local peer id: PeerId("12D3KooWC95ziPjTXvKPNGoz3CSe2yp6SBtKh785eT  
Listening on local address "/ip4/127.0.0.1/tcp/50996"
```

```
Listening on local address "/ip4/192.168.1.74/tcp/50996"
discovered 12D3KooWCVVb2EyxB1WdAcLeMuyaJ7nnfUCq45YNNuFYcZPGBY1f /
discovered 12D3KooWCVVb2EyxB1WdAcLeMuyaJ7nnfUCq45YNNuFYcZPGBY1f /
discovered 12D3KooWNgYbVg8ZyJ4ict2N1hdJLKoydB5sTqwiWN2SHtC3HwWt /
discovered 12D3KooWNgYbVg8ZyJ4ict2N1hdJLKoydB5sTqwiWN2SHtC3HwWt /
```

Notice that *node3* has discovered both *node1* listening on port 50960 and *node2* listening on port 50967.

This looks trivial, until you realise that we have not told *node3* where the other two nodes are running. Using the *mDNS* protocol, *node3* was able to detect and connect to other libp2p nodes on the local network.

11.5 Summary

- In client-server model of computation, the client and server represent two distinct pieces of software wherein the server is the custodian of data and associated computation, and the client requests the server to send data or perform a computation on a resource managed by the server. In P2P networks, communication occurs between peer nodes, each of which can perform the role of both the client and the server. One key characteristic that differentiates *client-server* networks from *P2P networks* is the absence of dedicated *servers* that have unique privileges
- *libp2p* is a modular system of protocols, specifications and libraries that enable development of peer-to-peer applications. It is used in many prominent p2p projects. Key architectural components of *libp2p* include transport, identity, security, peer discovery, peer routing, content routing and messaging.
- Using code examples, we looked into how to generate a unique peer id for a node that other nodes can use to uniquely identify it.
- We also delved into the basics of *multiaddresses*, and how they represent the complete path to communicate with a node over the P2P network. The *peer id* of a node is a part of the overall *multiaddress* of the node.
- We wrote a Rust program where nodes exchange simple ping-pong commands among themselves. This example demonstrated how to configure the swarm network management object for a node to listen and act on specific events on the p2p network.

- We concluded the chapter by writing another Rust program with the *libp2p* library that shows how peer nodes can use the mDNS protocol to discover each other on a p2p network.

For readers looking for additional code challenges, here are a couple of P2P applications that can be built using *libp2p*:

1. Implement a simple p2p chat application
2. Implement a distributed p2p key-value store.
3. Implement a distributed file storing network (like IPFS)

Hint: The libp2p library has several pre-built code examples which can be referred to, in order to implement these exercises. Reference to the code repository is provided at the end of this chapter.

With this, we come to an end of this chapter, and also this section on advanced topics. In the next (and last) chapter, we'll learn how to prepare Rust servers and apps for production deployment.

See you in the next chapter!

11.6 References

This chapter has heavily drawn material from the libp2p documentation at <https://libp2p.io/>. The code examples use the Rust implementation of the libp2p protocol which can be found here: <https://github.com/libp2p/rust-libp2p>.

12 Deploying web services with Docker

This chapter covers

- Introduction to production deployment of Rust servers and apps
- Writing the first Docker container
- Building the database container
- Packaging the web service with Docker
- Orchestrating Docker containers with Docker Compose

In earlier sections of the book, we learned how to build a *web service* and a *web application* using Rust. We tested these in a local development environment. This is only the first step. The ultimate goal is usually to deploy in a production environment. Production deployment involves many aspects that are outside of the scope of this book, such as selection of an infrastructure provider, packaging the software, configuring secrets, adding configurable logs for monitoring and debugging, adding application-level security to the web service API endpoints, adding server-level security (with TLS, CORS), protecting secrets such as access credentials and keys, configuring monitoring tools and alerts, adding database backups. and a lot more. It is not the intent of this book to provide an exhaustive guide to all the considerations in preparing and deploying an application or service into production, or to enumerate the best practices in this regard. This is because this is not a Rust-specific topic, and also because there is a lot of publicly available material (and other books) that cover this topic very well.

Here, we will focus only on packaging the software, using a popular technique, called "containerization". It is one of the popular (and now largely mainstream) methods of production deployment. It involves packaging the application components and its dependencies in a container. The container can be deployed on multiple environments including the *cloud*.

In this chapter, we'll take a detailed look at the steps needed to *containerize*

the *Rust web service*. Once the *web service* is available as a *Docker container*, it is no different from a web service or application written in any other programming language from a production deployment standpoint, and all the standard guidelines and options to deploy Docker containers would apply.

Let's get started with a broad overview of the production deployment lifecycle.

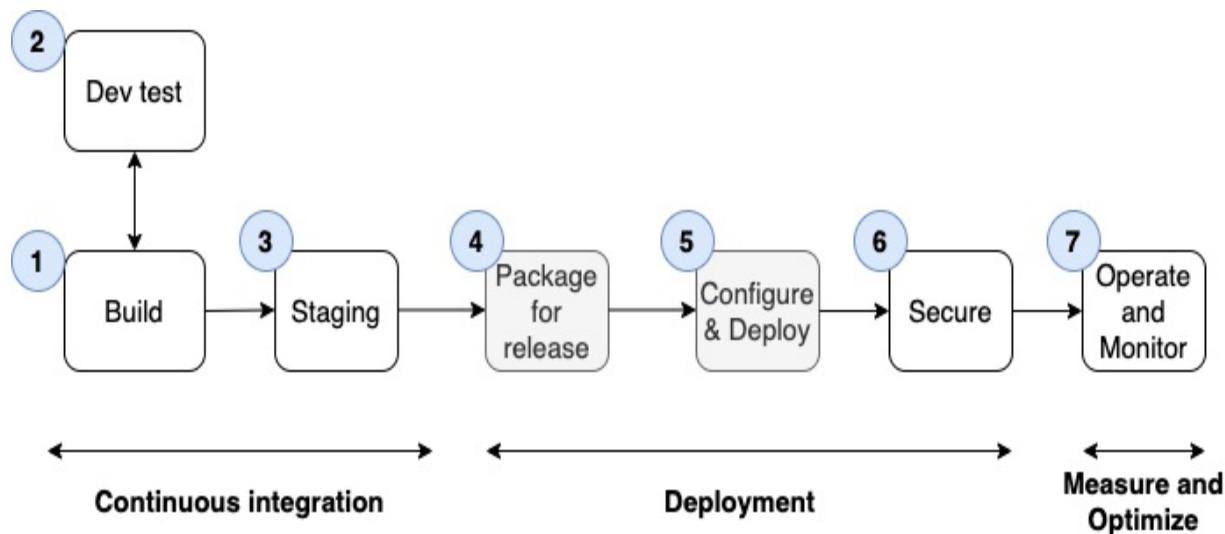
12.1 Introduction to production deployment of servers and apps

12.1.1 Software deployment cycle

The software deployment cycle involves multiple levels of developer unit and integration testing followed by preparation and deployment of the release. Once the release is deployed and running, the system is monitored, key parameters are measured and optimization is performed.

While the specific steps in the production deployment lifecycle varies by team and DevOps technologies, figure 12.1 shows a representative set of steps that are typically performed.

Figure 12.1. Production deployment lifecycle



Let's take a look at the various stages. The actual development steps and terminology used by various organisations differ widely, however let's use the following to gain a common conceptual understanding in broad strokes:

1. **Build**: Software is written (or modified) and the *binary* is locally built by the developer(s). Note that in most cases, this would be a *development build* (which facilitates debugging and takes lesser time to build), as opposed to a *production build* (which optimizes binary size but typically takes longer to build in most programming languages).
2. **Dev test**: The developers perform *unit tests* in a local development environment.
3. **Staging**: The code is merged with the other branches that are planned as part of a software release, and deployed in a *staging environment*. Here *integration tests* are performed involving code and modules written by other developers.
4. **Package for release**: After successful integration tests, the final *production build* is constructed. The method of packaging will involve decisions on how the binary will be deployed (e.g as a stand-alone binary, or deployed in a container or a public cloud service). **In this chapter we will focus on how to create a *Docker build* for the *Rust web service*.**
5. **Configure and deploy**: The production binary file is then deployed to the target environment (eg. virtual machine), and the needed configuration and environmental parameters are set up. This is also the stage where connection to additional components in the production infrastructure is performed. For example, the binary may be required to work with load balancers or reverse proxies. **In this chapter we will use *Docker Compose* to streamline the process of configuring, automating builds, starting and stopping the set of Docker containers needed to run the web service.**
6. **Secure**: It is here that additional security requirements are configured such as authentication (e.g. for user and API authentication), authorization (setting up user and group permissions) and network and server security (e.g. firewalls, encryption, secrets storage, TLS-termination, certificates, CORS, IP port enabling etc).
7. **Operate and Monitor**: This is where the server/binary is started in order to receive network requests, and the performance of the server is

monitored using network, server, application and cloud-monitoring tools. Examples of such tools include Nagios, Prometheus, Kibana and Grafana, to name a few.

In organizations where DevOps tools are deployed, *continuous integration*, *continuous delivery* and *continuous deployment* practices and tools are used to automate many of these steps. There is a plethora of publicly available material if you want to understand these terms in more detail.

In this chapter, we will focus only on a subset of these topics and show how to perform them in the context of the Rust programming language. We will specifically cover steps #4-*Package for release* and #5- *Configure and deploy*. For the latter, we will only focus on deploying the Docker containers on a Linux Ubuntu virtual machine, but the Docker containers can be deployed to any cloud provider (though there may be provider-specific steps needed for the deployment).

You will specifically learn the following:

- a) **Building the release binary and packaging:** You'll learn how to build the Rust server as a Docker image that can be deployed to any host with a container runtime. You'll learn how to write Dockerfiles, create Docker volumes and networks, configure environment variables, do multi-step Docker builds and reduce the size of final Docker images.
- b) **Configuring and deploying the web service:** You'll learn how to use Docker Compose to define runtime configuration of the web service and postgres database containers, define the dependencies between them, configure run-time environment variables, initiate Docker builds, and to start and stop the Docker containers through simple commands.

Let's start with a brief introduction to Docker.

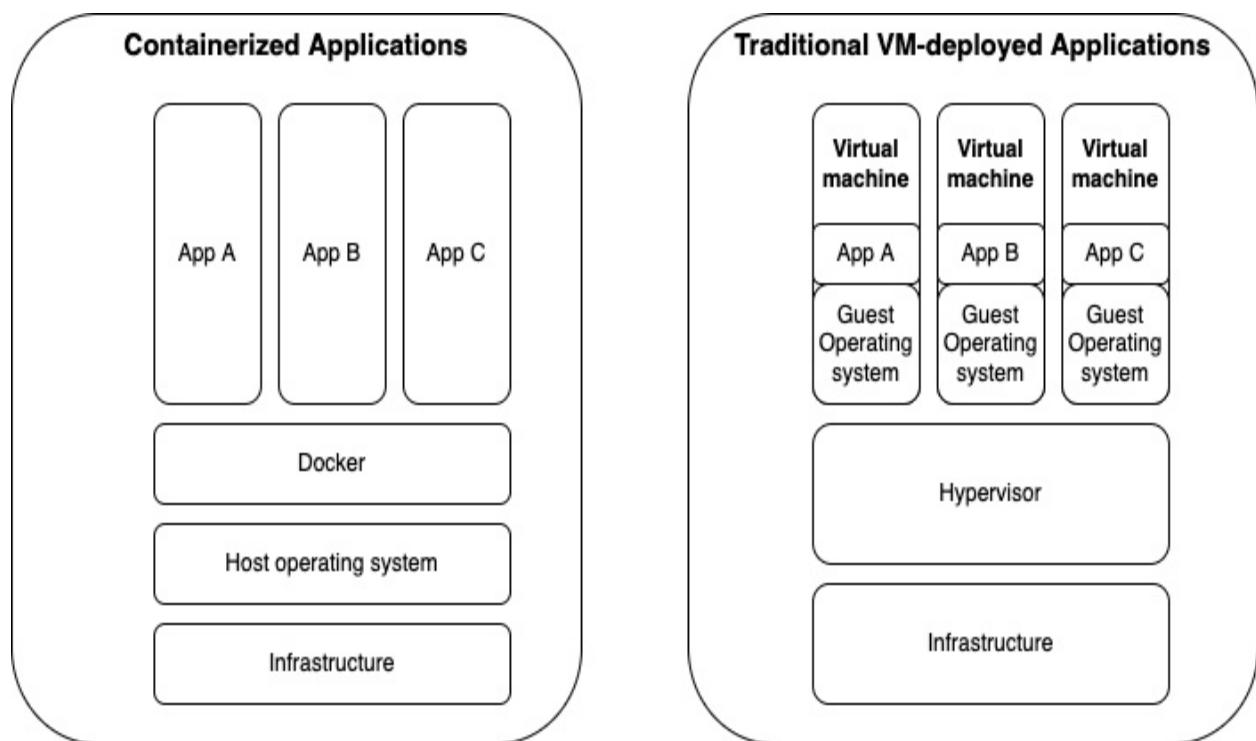
12.1.2 Docker container basics

Container technology has changed the way software is built, deployed and managed, enabling DevOps automation by bridging the gap between development and IT operation teams. *Docker* is both the *name of the*

company that played a major role in popularizing container technology, and also the name of the software product.

Docker containers are completely isolated environments with their own processes, networking interfaces and volume mounts. One important aspect of Docker containers is that they all ultimately share the same operating system kernel.

Figure 12.2. Containerized vs traditional VM-deployed Applications

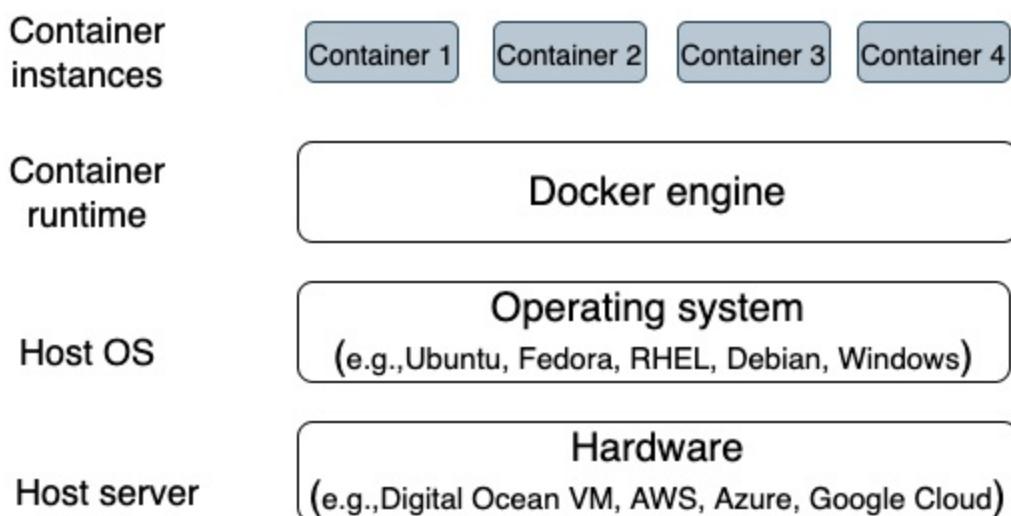


(Ref: <https://www.docker.com/resources/what-container/>)

Figure 12.2 shows the differences between a traditional VM-hosted vs containerized applications. Virtual machines are an *abstraction of physical hardware* turning one physical server into multiple 'logical' servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of the operating system. On the other hand, *containers are an abstraction at the application layer* that package code and dependencies together. Multiple containers run on the same physical machine and share the OS kernel with other containers. For more details refer to : <https://www.docker.com>.

Figure 12.3 shows a simple layered view of how Docker containers fit on the hardware infrastructure. Docker containers can contain any software application - a web service, a web application, a database or a messaging system, to name a few. Docker containers are lightweight (compared to VMs), can start up and shut down very quickly, and are self-contained in terms of the software application and all associated dependencies such as third-party crates and other libraries.

Figure 12.3. Docker container basics



One interesting aspect of Docker containers is that while the Docker host can be running the Ubuntu operating system, the Docker container can encapsulate the web service process running on Debian OS. This gives tremendous flexibility during development and deployment.

How does this facilitate the handshake between the software developers and the operations teams?

In traditional software deployment, the development team hands over the software components and associated configuration (in our case, the web service code repo, build instructions, instructions on prerequisites to be set up, postgres database scripts, environment files with secrets etc). The operations team will then have to follow the instructions to build and deploy the web service in the production environment. Developers are likely to build and test code in an environment different from the production environment.

The operations team, unfamiliar with the software, may run into issues which will require the presence of the development teams to resolve.

Docker containers solve this problem. Developers specify the infrastructure configuration, instructions to set up the environment, download and link the dependencies, and build the binary in a *Dockerfile*. The *Dockerfile* is a text file in *YAML* syntax. It allows one to specify parameters such as what is the base Docker image, environment variables to use, filesystem volumes to mount, ports to expose etc. The Dockerfile is then built into a customized Docker image, based on the rules specified in the *Dockerfile*.

The Docker image is the template from which multiple container runtimes can be instantiated. (The relationship between a *Docker image* and a *Docker container* is similar to the relationship between a *class* and an *object* in object-oriented programming languages.).

The developers instantiate the Docker image into Docker containers and test their software application. They then hand over the Docker image to the software operations team for a production deployment. Given that the Docker image is guaranteed to run the same way in any hardware infrastructure (Docker host), it becomes a lot easier for the operations teams to deploy and instantiate the software application in the production environment. Docker thus dramatically reduces the friction and human errors in production deployment of software applications. However, one flipside of using *Docker* is that it requires skilled Docker engineers to configure the build rules for an application.

More information about Docker can be found here:
<https://docs.docker.com/get-started/overview/>.

A prerequisite for this chapter is to install the Docker development environment in your development machine or server (Mac, Windows or Linux). See instructions here: <https://docs.docker.com/get-docker/>.

In the next section, we will write our first Docker container, and optimize its size.

12.2 Writing the first Docker container

In this section we will check the installation of Docker, write a *Dockerfile*, build the *Dockerfile* into a Docker image, and optimize the size of the final Docker image using a *multi-stage* build.

Let's start with checking the Docker installation.

12.2.1 Check Docker installation

You can create a project folder in your development server to follow-along with the code in this section.

From the terminal, check Docker installation with:

```
docker --version
```

You should get a response similar to the following one, on your terminal:

```
Docker version 20.10.16, build aa7e414
```

Let's test the official Docker image:

```
docker pull hello-world
```

You should see similar output:

```
Using default tag: latest
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:80f31da1ac7b312ba29d65080fddf797dd76acfb870e677f39
Status: Downloaded newer image for hello-world:latest
docker.io/library/hello-world:latest
```

Now check if the Docker image is available on your local dev server:

```
docker images
```

You should see the following:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
------------	-----	----------	---------	------

```
hello-world      latest      feb5d9fea6a5      8 months ago    13.3kB
```

You will see a Docker image *hello-world* available in your local dev server, with a Docker image id specified. Note also the size of the docker image. We'll talk later in the chapter about optimizing the size of Docker images.

As mentioned earlier, a Docker image is a template to create a Docker container instance. Let's instantiate the Docker image and see what happens:

```
docker run hello-world
```

If you see the following message, your Docker environment is good to go:

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub (amd64)
3. The Docker daemon created a new container from that image which contains an executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, writing it to your terminal.

To try something more ambitious, you can run an Ubuntu container:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

This official Docker image prints out the 'Hello from Docker!' message. That's all it does.

While using a Docker image created by someone else is useful, it is more interesting to create our own Docker image. Let's do that next.

12.2.2 Writing a simple Docker container

Start a new project with

```
cargo new --bin docker-rust  
cd docker-rust
```

This will be the project root folder.

Add Actix web to *Cargo.toml* dependencies:

```
[dependencies]  
actix-web = "4.2.1"
```

Add the following to *src/main.rs*:

```
use actix_web::{get, web, App, HttpResponse, HttpServer, Responde  
#[get("/")]  
async fn gm() -> impl Responder {  
    HttpResponse::Ok().body("Hello, Good morning!")  
}  
  
async fn hello() -> impl Responder {  
    HttpResponse::Ok().body("Hello there!")  
}  
  
#[actix_web::main]  
async fn main() -> std::io::Result<()> {  
    HttpServer::new(|| {  
        App::new()  
            .service(gm)  
            .route("/hello", web::get().to(hello))  
    })  
    .bind(("0.0.0.0", 8080))?  
    .run()  
    .await  
}
```

Let's first build and run the server in the regular mannet (without Docker):

```
cargo run
```

From the browser window, test the following:

```
localhost:8080
```

localhost:8080/hello

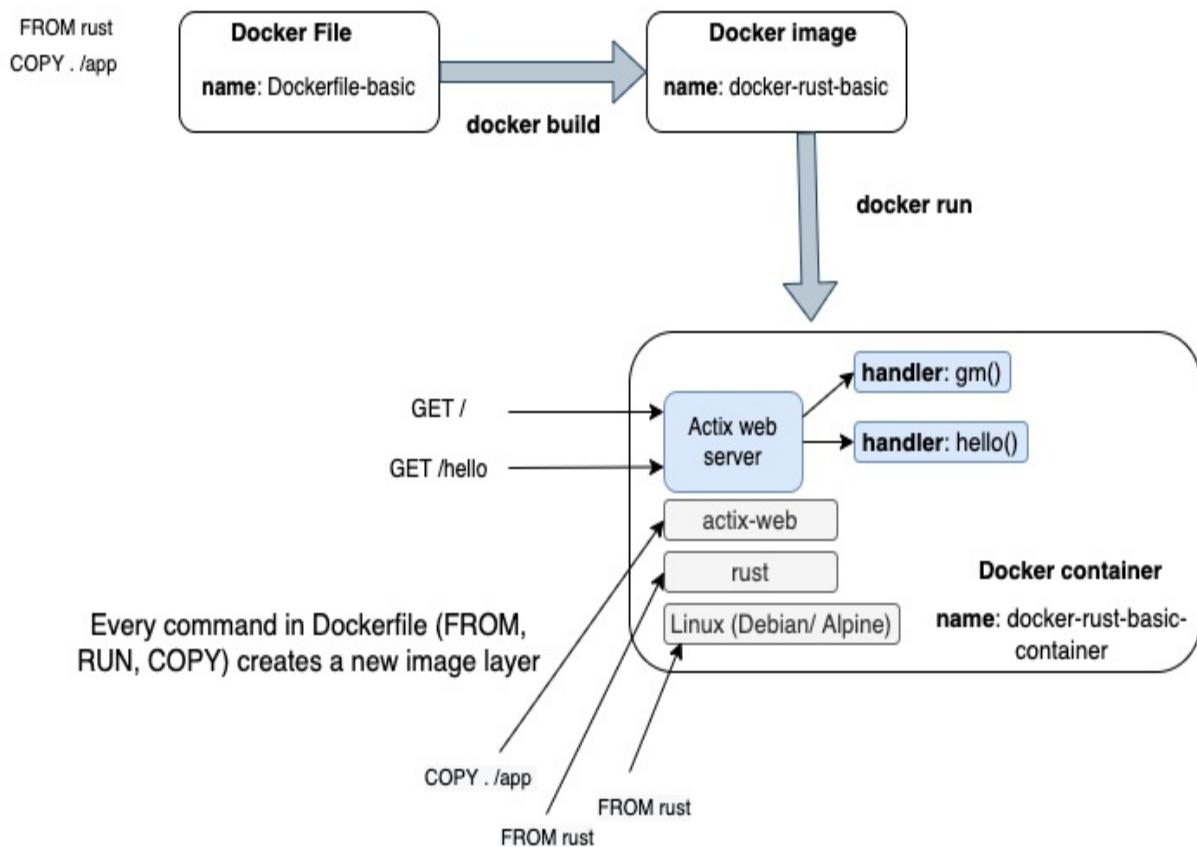
You should see the following messages (corresponding to the previous two GET requests) in the browser window:

Hello, Good morning!
Hello there!

Now that we've confirmed that the web service is working, let's *containerize* this web service with Docker.

Figure 12.4 shows what we will be building.

Figure 12.4. First Docker container



Create a new file *Dockerfile-basic* in the project root, and add the following:

```
# Use the main rust Docker image
FROM rust
```

```
# copy app into docker image
COPY . /app

# Set the workdirectory
WORKDIR /app

# build the app
RUN cargo build --release

# start the application
CMD ["./target/release/docker-rust"]
```

Run the following command to build the Docker image

```
docker build -f Dockerfile-basic . -t docker-rust-basic
```

You will see a series of messages ending with these:

```
=> => exporting layers
=> => writing image sha256:20fe6699b10e9945a1f0072607da46f726476
=> => naming to docker.io/library/docker-rust-basic
```

To check the docker image that has been built, run the following command:

```
docker images
```

You should see an output on your terminal similar to this:

REPOSITORY	TAG	IMAGE ID	CREATED	S
docker-rust-basic	latest	20fe6699b10e	9 seconds ago	1

You will notice that a Docker image with name *docker-rust-basic* has been created, with a specific Docker image id. The Docker image has a size of 1.32 GB. The reason is that Docker images include all the layers along with all their dependencies. For example, in this case, the Rust Docker image contains the Rust compiler and all the intermediate build artifacts which are not necessary to run the final application. But getting a large Docker image size in the first iteration is normal, as our initial priority is to get the Docker image defined and constructed the right way. We'll see later how to reduce the size of the Docker image.

Let's run the web server within this Docker container as shown:

```
docker run -p 8080:8080 -t docker-rust-basic
```

From the browser window, test the following:

```
localhost:8080  
localhost:8080/hello
```

You should see the respective messages displayed in the browser window.

We have now tested the web service in two versions: the basic version with *cargo run* and the *Dockerized* version.

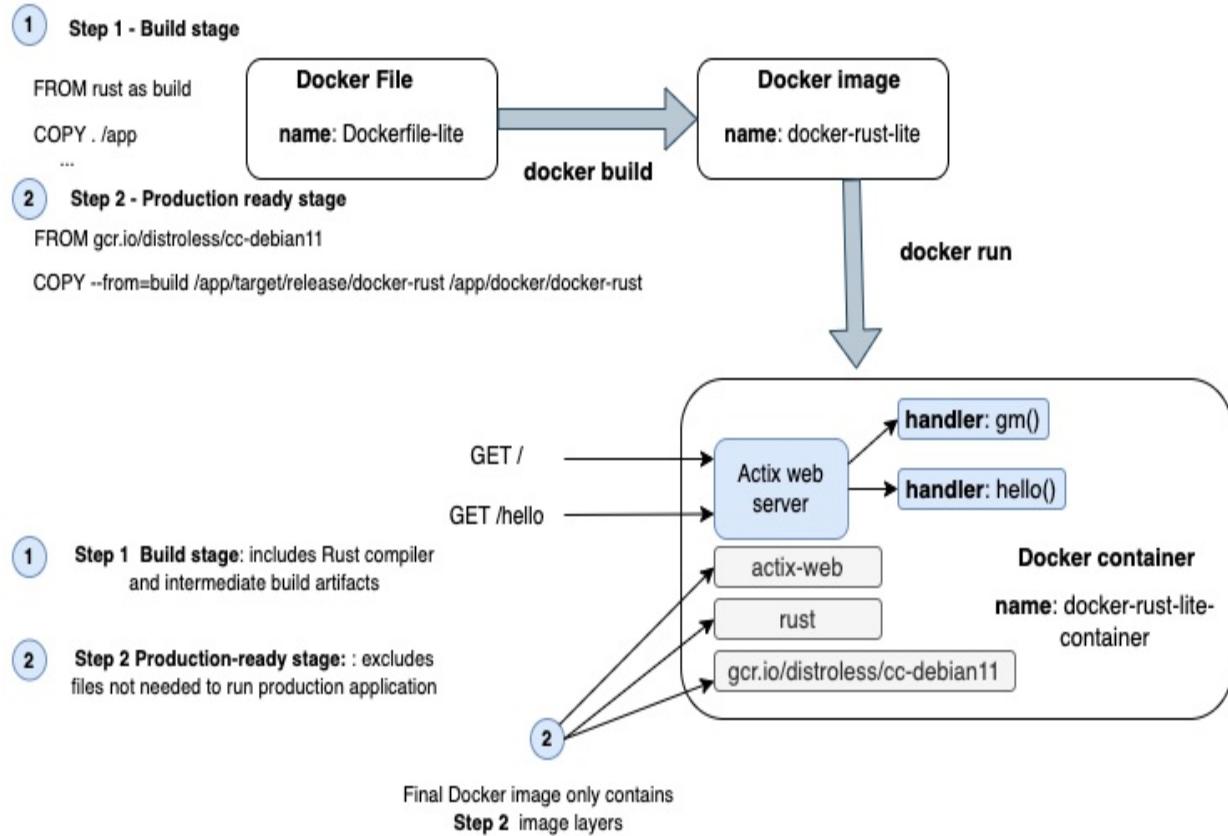
But we're not done yet. The problem we still have is that the Docker image of the web service has a size of 1.32 GB. Not exactly small. Docker binaries are expected to have a small footprint, but the Dockerized version of this very simple (and trivial) Rust web service has a large size. Can we fix it? Let's look at it in the next section.

12.2.3 Multi-stage Docker build

In this section, let's try to reduce the size of the Docker image.

Figure 12.5 shows what we will be doing in this section.

Figure 12.5. Lite Docker container



Create a new Dockerfile - *Dockerfile-lite* in the project root, and add the following:

```
# Use the main rust Docker image
FROM rust as build

# copy app into Docker image
COPY . /app

# Set the workdirectory
WORKDIR /app

# build the app
RUN cargo build --release

# use google distroless as runtime image
FROM gcr.io/distroless/cc-debian11

# copy app from builder
COPY --from=build /app/target/release/docker-rust /app/docker-rus
WORKDIR /app
```

```
# start the application
CMD ["./docker-rust"]
```

Run the following command to build the Docker image

```
docker build -f Dockerfile-lite . -t docker-rust-lite
```

To check the Docker image that has been built, run the following command:

```
docker images
```

You should see an output on your terminal similar to this:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZ
docker-rust-lite	latest	40103591baaf	12 seconds ago	31.

You'll now notice that the size of the Docker image has reduced to 31.8 MB.

Before we analyze it, let's first confirm that this Docker image actually works. Run the Docker image with the following command:

```
docker run -p 8080:8080 -t docker-rust-lite
```

Check the running container with:

```
docker ps
```

You should see the container *docker-rust-lite* shown in the list.

From the browser window, test the following:

```
localhost:8080
localhost:8080/hello
```

You should see the respective greeting messages displayed in the browser window.

So, how did this work?

We used what is called a *multi-stage* build. A *multi-stage Docker build* is a series of steps to create a Docker image. The main benefit of a multi-stage

build is to clean up after a development build and reduce the size of the final binary by removing extraneous files in the final Docker image. It lets developers automate the process of creating several versions of a binary aimed at different target OS environments, and also offers security and caching benefits.

A *Docker multi-stage build* uses several FROM statements to reference a specific image for that stage. Each stage can be named using the AS keyword. In the *Dockerfile-lite* example shown previously, we have two stages. The first stage of build builds a release binary. The second stage of build uses *google distroless* as a runtime image, and copies over the release binary previously created, which results in a smaller Docker image size.

Figure 12.6. Multi Stage builds

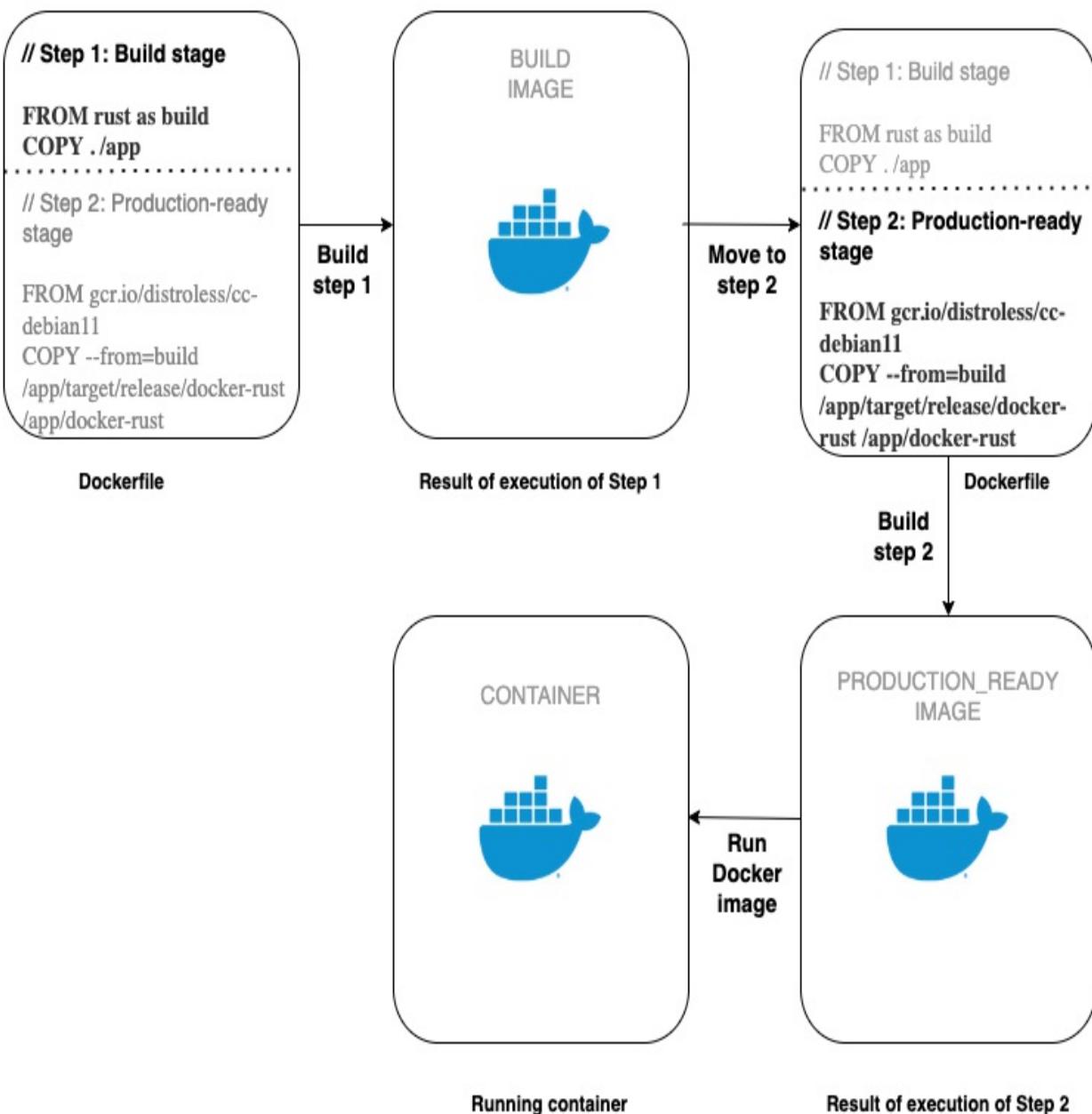


Figure 12.6 shows an example of a Docker multi-stage build with two steps. A single Dockerfile defines two build steps. The first build step creates a dev build docker image which contains dev-related artifacts. The second build step builds a production-ready Docker image which achieves a smaller size by excluding unwanted files.

More details on multi-stage Docker builds can be found here:
<https://docs.docker.com/develop/develop-images/multistage-build/>.

To summarize, the main difference between what's shown in Fig 12.4 and Fig 12.5 is that in the latter we have built the docker image in two steps, with the second (final) step excluding all the development tools and artifacts in the final Docker image.

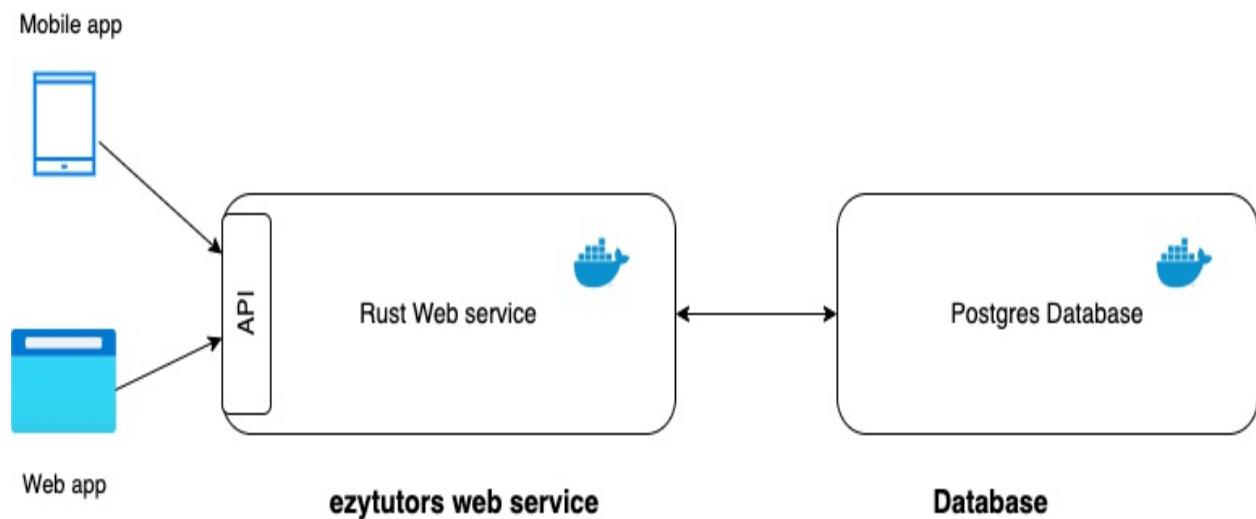
Now that we understand how to build and optimize a basic Rust actix program with Docker, let's shift our focus to the *ezytutors* web service.

12.3 Building the database container

The *ezytutors* web backend has two distinct components - the *web service* serving the APIs and the *postgres database*.

Figure 12.7 shows a visual representation of how we want to package the two components as docker containers, and have mobile and web clients send requests.

Figure 12.7. Docker Compose configuration



Let's first Dockerize the postgres database. We'll talk about how to package the *ezytutors* web service as a container in the next section.

But is there any real benefit of packaging the database as a Docker container?

Yes, because we want the *database* to be easily portable across machines,

and not tied to a specific hardware environment. We also eventually want to be able to operate (start, stop etc) the database and the web service together as one unit, which makes it easier if the database is also packaged as a container.

Let's get started.

12.3.1 Packaging the Postgres database

First clone the git repo for the book. Navigate to *chapter6/ezytutors/tutor-db*. This is the project root folder for the web service.

- a) Install Docker Compose on the Ubuntu server (or your own OD flavour). You may refer to documentation here:
<https://docs.docker.com/compose/install/>. The command to verify it on Ubuntu is : *docker compose version*. You should see an output similar to this:

```
Docker Compose version v2.5.0
```

- b) Create a new Docker network to interconnect the *tutor web service* and the *postgres database* containers.

```
docker network create tutor-network  
docker ls
```

You should see something similar to this:

6fc670fb70ba	bridge	bridge	local
75d560b02bbe	host	host	local
7d2c59b2f3a5	none	null	local
e230e1a9c55d	tutor-network	bridge	local

- c) Create a Docker volume.

Docker volumes are the preferred way to persist data generated by and used by Docker containers. They are completely managed by Docker. They are easy to backup and, using volume drivers, allow you to store data on remote hosts or cloud providers. A volume's contents exist outside the lifecycle of a Docker container. More details can be found here:

<https://docs.docker.com/storage/volumes/>.

Let's create a Docker volume as shown:

```
docker volume create tutor-data  
docker volume ls
```

You should see an output like this:

DRIVER	VOLUME NAME
local	tutor-data

d) Stop the postgresql database instance if running on the *Docker host*:

```
systemctl status postgresql  
systemctl stop postgresql
```

e) Create a new *Docker Compose* file with the name *docker-compose.yml*.
Add the following:

```
version: '3'  
services:  
  db:  
    container_name: tutor-postgres  
    restart: always  
    image: postgres:latest  
    environment:  
      - POSTGRES_USER=postgres  
      - POSTGRES_PASSWORD=postgres  
      - POSTGRES_DB=ezytutors  
    volumes:  
      - tutor-data:/var/lib/postgresql/data  
      - ./c12-data/initdb.sql:/docker-entrypoint-initdb.d/initdb.  
      - ./c12-data/init-tables.sql:/docker-entrypoint-initdb.d/in  
    ports:  
      - 5432:5432  
    networks:  
      - tutor-network  
  volumes:  
    tutor-data:  
  networks:  
    tutor-network:
```

With reference to annotation <1>, each entry under the *services:* keyword represents a separate docker container. In this case, we're telling Docker Compose that *db* is the name of the service and that a separate Docker

container should be spun up for the *db* service.

With reference to annotation <5>, the volume *tutor-data* on the Docker host is mapped to */var/lib/postgresql/data* (default database folder of postgres) within the Docker container.

With reference to annotation <6> *initdb.sql* contains the database scripts to create the database and users, and grant permissions.

With reference to annotation <7>, *init-tables.sql* contains the database scripts to create the database tables and load initial test data.

f) Build and run the postgres Docker image.

```
docker compose up -d  
docker ps
```

You should see an output similar to this:

CONTAINER ID	IMAGE	COMMAND	CREATED
d43b6ae99846	postgres:latest	"docker-entrypoint.s..."	4 secon
STATUS	PORTS		NAMES
Up 1 second	0.0.0.0:5432->5432/tcp, :::5432->5432/tcp		tutor

The Docker container *tutor-postgres* has been instantiated from the Docker image *postgres:latest*.

Let's check if the database and tables have been created and if test data has been loaded. For this, connect to the Docker container:

```
docker exec -it d43b6ae99846 /bin/bash #1  
psql postgres://postgres:postgres@localhost:5432/ezytutors #2  
\list #3
```

You should see a terminal output similar to this:

```
psql (12.11 (Ubuntu 12.11-0ubuntu0.20.04.1), server 14.3 (Debian  
WARNING: psql major version 12, server major version 14.  
          Some psql features might not work.  
Type "help" for help.
```

```

ezytutors=# \list
                                         List of databases
  Name   |  Owner   | Encoding | Collate    | Ctype    | Access p
-----+-----+-----+-----+-----+-----+
ezytutors | postgres | UTF8    | en_US.utf8 | en_US.utf8 |
postgres  | postgres | UTF8    | en_US.utf8 | en_US.utf8 |
template0 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgre
template1 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | postgres=C
(4 rows)

```

You should see the database *ezytutors* listed. This is because we placed the *initdb.sql* within the Docker container, in the folder /docker-entrypoint-initdb.d. Any script placed within this folder should be automatically executed when the container starts.

Just enter *\q* and exit the psql shell, followed by *exit* in the Docker bash shell, to exit the Docker container.

There is another way to access the database, which is to connect to the Docker container and execute psql from within it, as shown:

```

docker ps
docker exec -it 0027d5c1cfaf /bin/bash
psql -U postgres
\list

```

You should see such an output:

```

bash-5.1# psql -U postgres
psql (11.16)
Type "help" for help.

```

```

postgres=# \list
                                         List of databases
  Name   |  Owner   | Encoding | Collate    | Ctype    | Access p
-----+-----+-----+-----+-----+
ezytutors | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =Tc/postgr
           |          |          |             |          | postgres=C
           |          |          |             |          | truuser=CT
postgres  | postgres | UTF8    | en_US.utf8 | en_US.utf8 |
template0 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgre
           |          |          |             |          | postgres=C

```

```
template1| postgres| UTF8      | en_US.utf8| en_US.utf8| =c/postgre
          |           |           |           |           | postgres=C
(4 rows)
```

You'll notice that both are acceptable ways to access the `postgres` database within the `tutor-postgres` container.

We see that the `ezytutors` database has been created.

Let's check if the user `truuser` has been created and privileges assigned to the user. From within the Docker container, execute the following command at the command prompt:

```
psql -U truuser ezytutors
ezytutors=>\list
```

If you are able to see `ezytutors` database listed, it's good. Otherwise, execute the following steps:

Let's run these commands within the `psql` shell:

```
postgres=# drop database ezytutors
postgres=# \list
```

You should see a similar output:

```
postgres=# drop database ezytutors;
DROP DATABASE
postgres=# \list
                                         List of databases
   Name    |  Owner   | Encoding | Collate   | Ctype    | Access p
-----+-----+-----+-----+-----+-----+
postgres | postgres | UTF8    | en_US.utf8| en_US.utf8| =c/postgre
template0| postgres | UTF8    | en_US.utf8| en_US.utf8| =c/postgre
template1| postgres | UTF8    | en_US.utf8| en_US.utf8| =c/postgre
          |           |           |           |           | postgres=C
(3 rows)
```

We have deleted the database `ezytutors` because we want to execute the `initdb.sql` script in its entirety once again.

Now let's run the two initialization scripts that we stored within the Postgres

Docker container under *docker-entrypoint-initdb.d*.

Go back to the Docker container bash shell (not psql shell), and execute the following commands:

```
postgres=# \i /docker-entrypoint-initdb.d/initdb.sql
```

You should see a similar output in your terminal:

```
postgres=# \i /docker-entrypoint-initdb.d/initdb.sql
CREATE DATABASE
CREATE ROLE
GRANT
ALTER ROLE
ALTER ROLE
```

The *initdb.sql* script creates the database, creates a new user *truuser*, and grants all permissions to this new user on the database *ezytutors*.

Now quit the *psql* shell with *\q* and log back in from the Docker container *bash shell* with the *truuser* id as shown:

```
psql -U truuser ezytutors
ezytutors=>\list
```

You should see the following on your terminal:

```
ezytutors=> \list
                                         List of databases
   Name    |  Owner   | Encoding | Collate   |  Ctype   | Access privi
   ----+-----+-----+-----+-----+-----+
ezytutors | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =Tc/postgres
           |          |          |            |            | postgres=CTc
           |          |          |            |            | truuser=CTc/
postgres  | postgres | UTF8    | en_US.utf8 | en_US.utf8 |
template0 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgres
           |          |          |            |            | postgres=CTc
template1 | postgres | UTF8    | en_US.utf8 | en_US.utf8 | =c/postgres
           |          |          |            |            | postgres=CTc
(4 rows)
```

The *ezytutors* database can now be accessed by *truuser*. In the next section, we'll look at how to create database tables within the Docker container.

12.3.2 Creating database tables

From the command prompt of the Postgres Docker container, check if the database tables have been created using:

```
ezytutors=> \d  
Did not find any relations.
```

If you see the list of tables, then it's all good. But if you see the error message above *Did not find any relations*, then we need to manually run the script to create tables and load test data.

Let's now create the *tutor* and *course* related tables in *ezytutors* database, and then list the database tables (called relations in *postgres* language). We'll do this by executing the *init-tables.sql* script.

You should see this:

```
ezytutors=> \i /docker-entrypoint-initdb.d/init-tables.sql  
psql:/docker-entrypoint-initdb.d/init-tables.sql:4: NOTICE:table  
DROP TABLE  
psql:/docker-entrypoint-initdb.d/init-tables.sql:5: NOTICE:table  
DROP TABLE  
CREATE TABLE  
CREATE TABLE  
GRANT  
GRANT  
INSERT 0 1  
INSERT 0 1  
INSERT 0 1  
INSERT 0 1  
ezytutors=> \d  
          List of relations  
 Schema |           Name            |   Type    | Owner  
-----+-----+-----+-----  
 public | ezy_course_c6           | table    | truuser  
 public | ezy_course_c6_course_id_seq | sequence | truuser  
 public | ezy_tutor_c6             | table    | truuser  
 public | ezy_tutor_c6_tutor_id_seq | sequence | truuser  
(4 rows)
```

The tables have been created. Let's also check if the initial test data has been loaded into *tutor* and *course* tables:

```

ezytutors=> select tutor_id, tutor_name, tutor_pic_url from ezy_tutor
            tutor_id | tutor_name |          tutor_pic_url
-----+-----+-----+
      1 | Merlene    | http://s3.amazonaws.com/pic1
      2 | Frank      | http://s3.amazonaws.com/pic2
(2 rows)

ezytutors=> select course_id, tutor_id, course_name, course_format, course_level, from ezy_course_c6;

course_id | tutor_id | course_name | course_format | course_level
-----+-----+-----+-----+-----+
      1 |        1 | First course |           | Beginner
      2 |        2 | Second course | ebook       |
(2 rows)

```

All good so far.

It is now time to conduct a test. What happens when we stop the container?. Will the data persist between container restarts?

For this, let us add a new record to the tutor table, shut down the container and restart it to check if the data has persisted.

```

ezytutors=> insert into ezy_tutor_c6 values(3, 'Johnny', 'http://s
ezytutors=> \q
exit
root@1dfd3bd87e2c:/# exit

```

Exit the psql shell with *q*, and then issue the *exit* command on the bash shell of the Docker postgres container. This should take you to your project home folder.

Now shut down the Docker container with:

```

docker compose down
docker ps

```

Your postgres container should no longer be running.

Now restart the container, and get into the running container shell:

```
docker compose up -d
docker ps
docker exec -it 7e7c11273911 /bin/bash
```

Then, in the container, login to the database with *psql* client, and check that the *tutor* table has the additional entry that you added previously:

```
root@7e7c11273911:/# psql -U truuser ezytutors
psql (14.3 (Debian 14.3-1.pgdg110+1))
Type "help" for help.
```

```
ezytutors=> \d
                                         List of relations
 Schema |           Name            |   Type   |  Owner
-----+-----+-----+-----+
 public | ezy_course_c6          | table    | truuser
 public | ezy_course_c6_course_id_seq | sequence | truuser
 public | ezy_tutor_c6           | table    | truuser
 public | ezy_tutor_c6_tutor_id_seq | sequence | truuser
(4 rows)

ezytutors=> select * from ezy_tutor_c6;
 tutor_id | tutor_name | tutor_pic_url | tutor_desc
-----+-----+-----+-----+
      1 | Merlene   | http://s3.amazonaws.com/pic1 | Merlene
      2 | Frank     | http://s3.amazonaws.com/pic2 | Frank is
      3 | Johnny    | http://s3.amazonaws.com/pic2 | Johnny is
(3 rows)
```

The data has indeed been persisted.

We have now completed the task to create a postgres database container , initialize the database, and load test data. This concludes the setup of the Docker postgres container.

As the next step, we can now move on to *Dockerizing* the *tutor* web service in the next section.

12.4 Packaging the web service with Docker

In the previous section, we packaged the *ezytutors* *postgres database* as a Docker container. In this section, let us turn our attention to packaging the

tutor web service as a Docker container.

We will first create a *Dockerfile*. This is because we want to create a custom Docker image for the tutor web service (as opposed to using the standard postgres image in the previous section). The custom Dockerfile is required because of two reasons:

- There is no standard Docker image available in Docker hub for our tutor web service. This is our custom code, and we need to give instructions in Dockerfile to package it as a container
- We want to specify instructions to create a static self-contained binary , without the use of shared libraries. By default the Rust standard library dynamically links to the system *libc* implementation. Since we want a 100% static binary for the web service, we will use musl *libc* on the Linux distribution we use within the web service Docker container.

Why use RUST with MUSL?

By default, Rust statically links all Rust code. But if you use the standard library (which we do in this book), it will dynamically link to the system *libc* implementation. Operating system differences can cause Rust binaries to break when run in a different environment compared to that they were compiled in. For example, if the binary was built using newer version of Glibc compared to the target system (where the Rust program is deployed and run), it will fail to run. One of the approaches to avoid this problem is to statically compile MUSL into the binaries.

MUSL is a lightweight replacement for Glibc used in Alpine Linux. When MUSL is statically compiled into your Rust program, you can create a self-contained executable that will run without dependencies on Glibc. Credits: <http://mng.bz/44ra> This is the approach we will use in this book, to package Rust in Docker containers.

Let us first create the Dockerfile for the tutor web service.

Create a Dockerfile named *Dockerfile-tutor-webservice*, and add the following:

```

# Use the main rust docker image
FROM rust as build          #1
RUN apt-get update && apt-get -y upgrade #2
RUN apt-get install libssl-dev      #2
RUN apt-get -y install pkg-config musl musl-dev musl-tools #2
RUN rustup target add x86_64-unknown-linux-musl           #3

# copy app into Docker image
COPY . /app                                         #4

# Set the workdirectory
WORKDIR /app                                       #5

# build the app
RUN cargo build --target x86_64-unknown-linux-musl --release --bi

CMD ["/./target/x86_64-unknown-linux-musl/release/iter5"]

```

We have created the Dockerfile. We can run the *Docker build* command directly on this Dockerfile. But we will do it in a different way. Let us see how in the next section.

12.5 Orchestrating Docker containers with Docker Compose

In this section, we will use Docker Compose to create a multi-container configuration for the ezytutors application.

Why use Docker Compose?

Docker Compose is a client-side tool that lets you run an application stack with multiple containers.

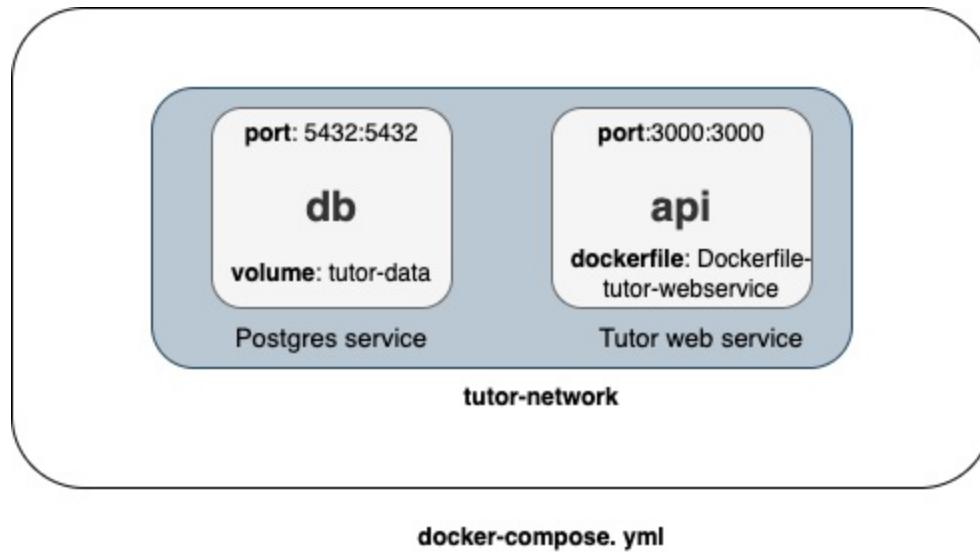
While *Docker* has made it easy to create local development environments for individual services, when there are multiple docker containers to manage for an application (as we have in our ezytutors example), it becomes cumbersome. Docker Compose solves this problem by specifying the configuration of one or more Docker containers within a single YAML configuration file.

Using Docker Compose, you can specify the build instructions, storage configuration, environment variables and network parameters for each Docker container that is part of a single application. Once defined, Docker Compose allows you to build, start and stop all the containers using a single set of commands.

Let us add tutor web service as a service within the *Docker Compose* file that we created in the previous section for the postgres database container. In this way, we have a single Docker Compose file that has details of both the Docker containers needed to build and run the tutor web service. Also, we can specify the dependencies between the two containers, and connect them through a common Docker network. Also, we can specify the Docker volume to which postgres data should be persisted between Docker container runs.

Figure 12.8 shows a visual representation of key elements of the final Docker Compose file for our application.

Figure 12.8. Docker Compose configuration



In *docker-compose.yml*, add *tutor-webservice* as a service. The complete Docker Compose yaml file should look like this:

```
version: '3'  
services:  
  db: #1
```

```

container_name: tutor-postgres
restart: always
image: postgres:latest
environment:
  - POSTGRES_USER=postgres
  - POSTGRES_PASSWORD=postgres
  - POSTGRES_DB=ezytutors
volumes:
  - tutor-data:/var/lib/postgresql/data
  - ./c12-data/initdb.sql:/docker-entrypoint-initdb.d/initdb.s
  - ./c12-data/init-tables.sql:/docker-entrypoint-initdb.d/ini
ports:
  - 5432:5432
networks:
  - tutor-network
api:                                     #2
  restart: on-failure
  container_name: tutor-webservice
  build:                                     #3
    context: ./
    dockerfile: Dockerfile-tutor-webservice
    network: host
  environment:                                #4
    - DATABASE_URL=${DATABASE_URL}
    - HOST_PORT=${HOST_PORT}
depends_on:
  - db                                         #5
ports:
  - ":3000:3000"                               #6
networks:
  - tutor-network                                #7
volumes:
  tutor-data:
networks:
  tutor-network:

```

We can now start the postgres database container with:

```
docker compose up db -d
```

This will start the database container alone, as a background process.

Before we build and run the tutor web service container, let us first check the environment variable settings.

```
cat .env
```

You should see this:

```
DATABASE_URL=postgres://truuser:trupwd@localhost:5432/ezytutors  
HOST_PORT=0.0.0.0:3000
```

Let us set the DATABASE_URL environment variable in the current terminal shell:

```
source .env  
echo $DATABASE_URL
```

You should see the value of database url correctly set as the environment variable. This step is important because *sqlx* does compile-time checking of the database, while building the tutor web service.

```
postgres://truuser:trupwd@localhost:5432/ezytutors
```

Let us double-check that the postgres url is accessible from the *Docker host* shell (to avoid unwanted delays in compilation process):

```
psql postgres://truuser:trupwd@localhost:5432/ezytutors  
\q
```

If this takes you to the postgres shell, you are ready to build the tutor web service container, as shown:

```
docker compose build api
```

It will take a while, depending on the configuration of your machine. So, go grab a coffee (or another drink of your choice).

Once the process is complete, check the built image with:

```
docker images
```

You should see this:

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
tutor-db_api	latest	23bee1bda139	52 seconds ago	2.87GB
postgres	latest	5b21e2e86aab	7 days ago	376MB

Now that we have built the *web service* container, let us start it up. But before

that we will have to shutdown the running postgres container, as the Docker Compose file will start both the *api* (web service container) and *db* (postgres container) services together.

Get the Docker image id and remove the running postgres container.

```
docker ps
docker stop <image id>
docker rm <image id>
```

Before we start the containers, there is one step to be done.

Recall that the *tutor web service* uses the *DATABASE_URL* environment variable to connect to the *postgres* database. While building the web service container, we set the following value to *DATABASE_URL*:

```
DATABASE_URL=postgres://trouser:trupwd@localhost:5432/ezytutors
```

Note that the value after the @ symbol represents the host on which the *postgres* database runs. During the build phase, we set it to *localhost*. But for the *tutor webservice* container (named as *api* service in Docker Compose file), *localhost* refers to itself. If so, how did it connect to the *postgres* container at build time? This is because we made a small hack at build time. If you look back at the Docker Compose file to build the *tutor web service*, you will notice the *network* parameter set to *host*.

```
api:
  restart: on-failure
  container_name: tutor-webservice
  build:
    context: .
    dockerfile: Dockerfile-tutor-webservice
    network: host
```

This parameter enabled the build process of the *tutor web service* container to proceed, by connecting to the *localhost port* of the *docker host* from which the Docker container build happened. But this is not suitable for a production environment. This is the reason we have created a separate Docker network called *tutor-network*, and specified that both containers are to be connected to this network. Let us verify it with:

```
docker network ls  
docker inspect tutor-network
```

If you do not see any reference to the *tutor web service* or *postgres* containers, then add them manually as shown:

```
docker network connect tutor-network tutor-webservice  
docker network connect tutor-network tutor-postgres  
docker inspect tutor-network
```

You should see a similar output:

```
"Containers": {  
    "26a5fc9ac00d815cb933bf66755d1fd04f6dca1efe1ffbc96f28da50e65238  
        "Name": "tutor-postgres",  
        "EndpointID": "e870c365731463198fbdf46ea4a7d22b3f9f497727b410  
        "MacAddress": "02:42:ac:1b:00:03",  
        "IPv4Address": "172.27.0.3/16",  
        "IPv6Address": ""  
    },  
    "af6e823821b36d13bf1b381b2b427efc6f5048386b4132925ebd1ea3ecfa5e  
        "Name": "tutor-webservice",  
        "EndpointID": "015e1dbc36ae8e454dc4377ad9168b6a01cae978eac4e0  
        "MacAddress": "02:42:ac:1b:00:02",  
        "IPv4Address": "172.27.0.2/16",  
        "IPv6Address": ""  
    }  
},
```

The two containers *tutor-postgres* and *tutor-webservice* have been added to the *tutor-network*.

Now within a network, the containers can access each other by the container names. So, the *tutor-webservice* can access the *postgres* container using the name *tutor-postgres*. Let us now modify the database url as shown, in the *.env* file:

```
DATABASE_URL=postgres://truuser:trupwd@tutor-postgres:5432/ezytut
```

Note that the host value is now set to *tutor-postgres* instead of *localhost*. Let us set the environment variable in the shell, and restart the containers.

```
source .env
```

```
echo $DATABASE_URL
echo $HOST_PORT
docker compose down      #1
docker compose up -d     #2
docker network connect tutor-network tutor-webservice #3
docker network connect tutor-network tutor-postgres   #3
docker inspect tutor-network    #4
```

Now from your server terminal (not inside Docker), run the following to check the web service endpoint:

```
curl localhost:3000/tutors/
```

You should see the following result:

```
[{"tutor_id":1,"tutor_name":"Merlene","tutor_pic_url":"http://s3.
```

Note that the additional entry you added to the list of tutors is also shown, confirming that the database changes are persisted to the local volume across container restarts. You can run tests on the other end points also as an exercise.

Congrats, if you have come this far. You have successfully Dockerized the tutor web service and the postgres database. You have also made the task greatly simpler by using *Docker Compose* to build, start and stop all the containers together with simple commands.

With this, we can conclude this chapter.

12.6 Suggested exercises

For readers looking for additional code challenges, here are a few:

1. Docker build commands can take a long time to create a Docker image. Explore usage of cargo chef (<https://github.com/LukeMathWalker/cargo-chef>) to speed up container builds
2. Add middleware to the Actix web server, which can be used to add additional functionality such as CORS, JWT authentication of API endpoints and logging levels. For more details see here:

<https://actix.rs/docs/middleware/>

3. The size of the tutor web service container image in the previous section is large ~ 2.87 GB. As an exercise, enhance the Dockerfile *Dockerfile-tutor-webservice* to include a multi-stage build and reduce the size of the Docker image. More details on multi-stage builds can be found here: <https://docs.docker.com/develop/develop-images/multistage-build/>.

12.7 Summary

- Rust web services, applications and databases can be packaged into Docker containers. Docker is a popular way to build and run light-weight containers that removes friction between the software developers and operations teams.
- Docker files contain the instructions to build the Docker image. From the image, containers can be instantiated which can service requests. For containerizing Rust programs, building static Rust binaries with MUSL helps avoid issues with *libc* versions on different target environments.
- Multi-stage Docker builds can be used to reduce the size of final Docker images. In case of Rust, the first stage involves installing the Rust development environment and associated dependencies to build the static Rust binary. The second stage involves removing the Rust compiler and intermediate build artifacts by creating a new base image and copying only the final Rust static (self-contained) binary.
- Docker containers can be grouped together using Docker Compose, a tool to build, run and manage the life cycle of a set of Docker containers together
- We defined two services (Docker containers) as part of the Docker Compose file - a postgres database and the tutor web service
- *docker build* , *docker images*, *docker run*, *docker ps*, *docker inspect*, *docker compose up*, *docker compose down*, and *docker compose build* are some of the commonly used Docker commands
- Since sqlx does compile-time checking of the database, the postgres container is started first, and then the tutor web service is built.
- Docker containers can be interconnected using a custom Docker network.
- Docker volumes can be used to persist data to disk between Docker container runs.

- Docker Compose greatly simplifies the lifecycle management of a group of containers
- Dockerfiles and Docker Compose files for a project can be used to deploy an application or service on various virtual infrastructure and cloud providers

With this, we come to an end of this chapter, and also this book.

This book is designed to get you started on the journey to writing web services and applications in Rust. But this is where I get off, and let you explore and enjoy the world of Rust web development on your own.

I wish you the best in your continued exploration of Rust servers, services and apps development.

Appendix A. Postgres installation

You may choose to install postgres in one of the following ways:

- Local installation on macOS, Windows or Linux/Unix development environment
- Run postgres database in a docker container
- Connect to a hosted and managed postgres database on the cloud such as AWS, Azure , google cloud, heroku or digital ocean

The instructions to install postgres on a Linux Ubuntu server is given here:

Refresh the local package index:

```
sudo apt update
```

Install the postgres package along with a *contrib* package that has additional utilities.

```
sudo apt install postgresql postgresql-contrib
```

Now the *postgres* software is installed. The installation also automatically starts the *postgresql* server as a *systemd* process in Linux. To verify this, type:

```
ps aux | grep postgres
```

You should see the *postgres* processes running in the background.

Let's now interact with the *postgres* database management system.

By default, *postgres* uses the concept of "role" (which is similar to users in Linux/Unix) to handle authentication and authorization. The installation process creates a user account called *postgres*. Log into the account as shown:

```
sudo -i -u postgres
```

You should now see the shell corresponding to *postgres* user;

From here, you can access the Postgres shell prompt , which allows us to interact with the postgres database management system to perform tasks such as creating database, creating users etc. Simply type:

```
psql
```

This will log you into a PSQL prompt.

You can exit out of the prompt anytime using:

```
\q
```

Now exit the postgres user prompt with

```
exit
```

Next, we need to make a change to the *postgres* configuration to allow *peer authentication*:

Look for *pg_hba.conf* file under */etc/postgres*. For example, for a postgres version 12 installation, this file can be found at:

```
/etc/postgresql/12/main/pg_hba.conf
```

Open the file in a text editor such as *vim* or *nano* and look for the following entry:

```
# "local" is for Unix domain socket connections only
local    all          all                                     p
```

Replace **peer** with **md5** as shown:

```
local    all          all                                     m
```

Save the file and restart the postgres server as follows:

```
sudo systemctl restart postgresql
```

This configuration change allows you to login to a postgres database with a password, once you are logged into the server.

Note also the following steps that need to be performed:

- * Create a database
- * Create a user and associate a password
- * Assign privileges for the user to the database

Once these steps have been done, you will be able to login to the postgres database from the command line, using:

```
psql -U <database-user> -d <database-name> --password
```

The *database-user* and *database-name* have to be replaced with your own. The --password flag will prompt for a password entry.

For more details, refer to postgres official documentation at www.postgresql.org/docs/.