Rust Servers, Services, and Apps

Prabhu Eshwarla

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Rust Servers, Services, and Apps
Version 1

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

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 elightful 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 inspite 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

brief contents

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-boggingly 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 executing 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 sociey will 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, fullstack 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 applications and modern WASM-based client front-ends 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 together build 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 airtraffic 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.

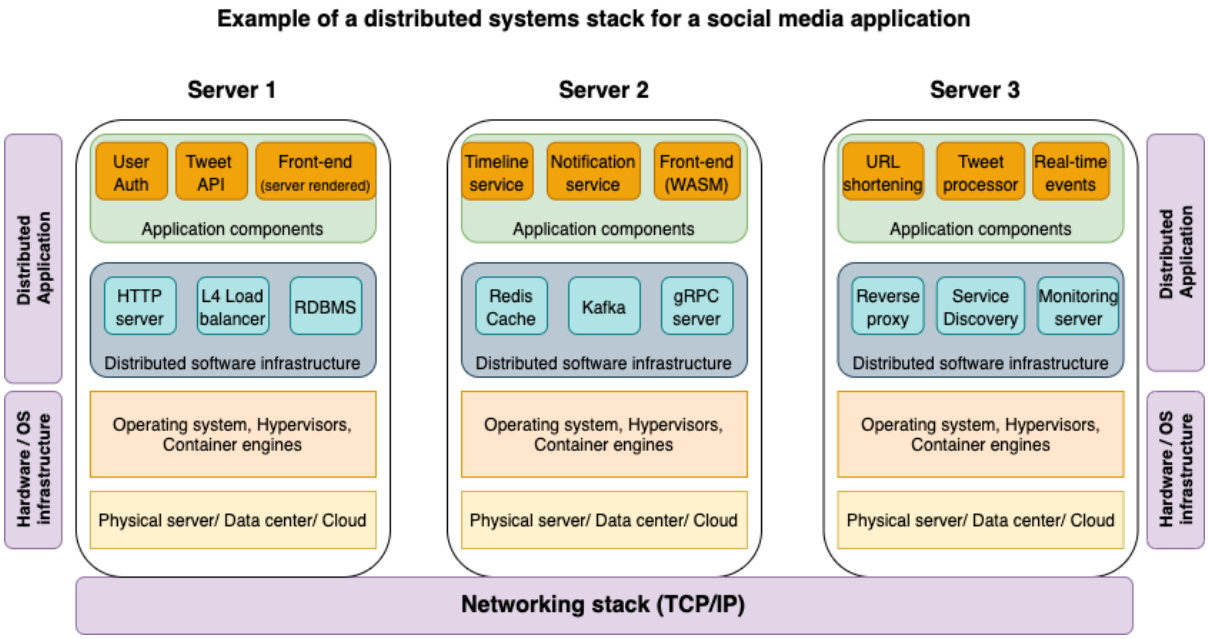

Figure 1.1 Distributed systems stack (simplified)

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 lighest 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 distrbuted 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 process 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 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 of components with atleast 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 something we will discuss in the chapter on developing WASM-based front-ends in Rust.

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 safety 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 of 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 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 more 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/.../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 has even been formalized as a software technique called horizontal scalability. But the real reason this has become popular is due to limitation 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 applicances 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. Adiitionally, *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/../rust/.

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 web site?*
>
> **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.

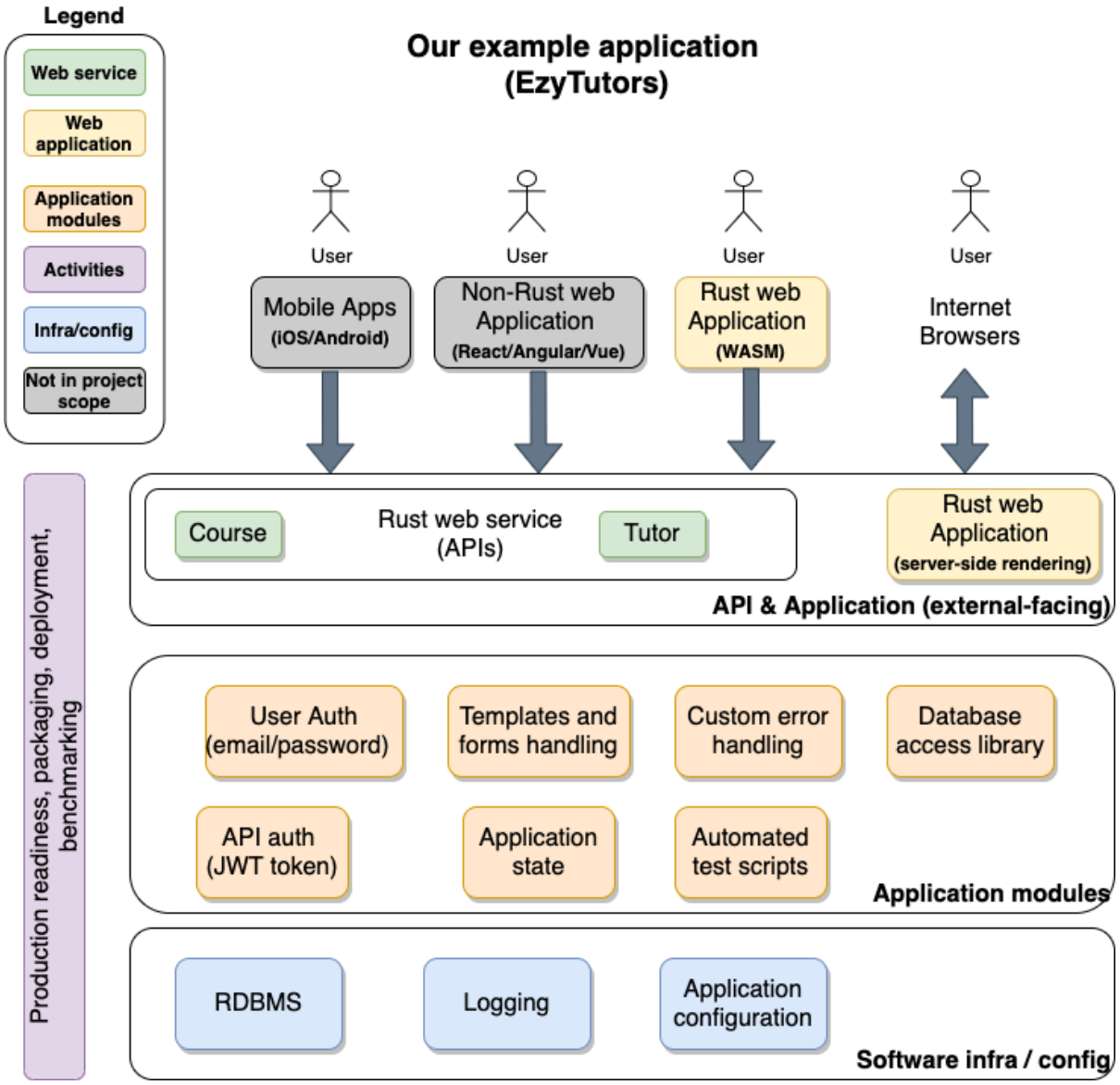

Figure 1.2 Our example application

Our technical stack will be comprised 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 two types of front-ends in Rust - a *server-rendered client app* and a *WASM-based in-browser app*. We'll use a template engine to render templates and forms for the *server-rendered web application*. We'll develop a *WASM-based client front-end* using a Rust client-side web framework. This will be a welcome relief from a *javascript-centric* front-end world.

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 libaries sometimes impedes the learning process and deep understanding.
7. **Async I/O**: It is a deliberate choice to use libaries 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 such as REST, gRPC, websockets and TLS 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 (commandline) 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/../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.

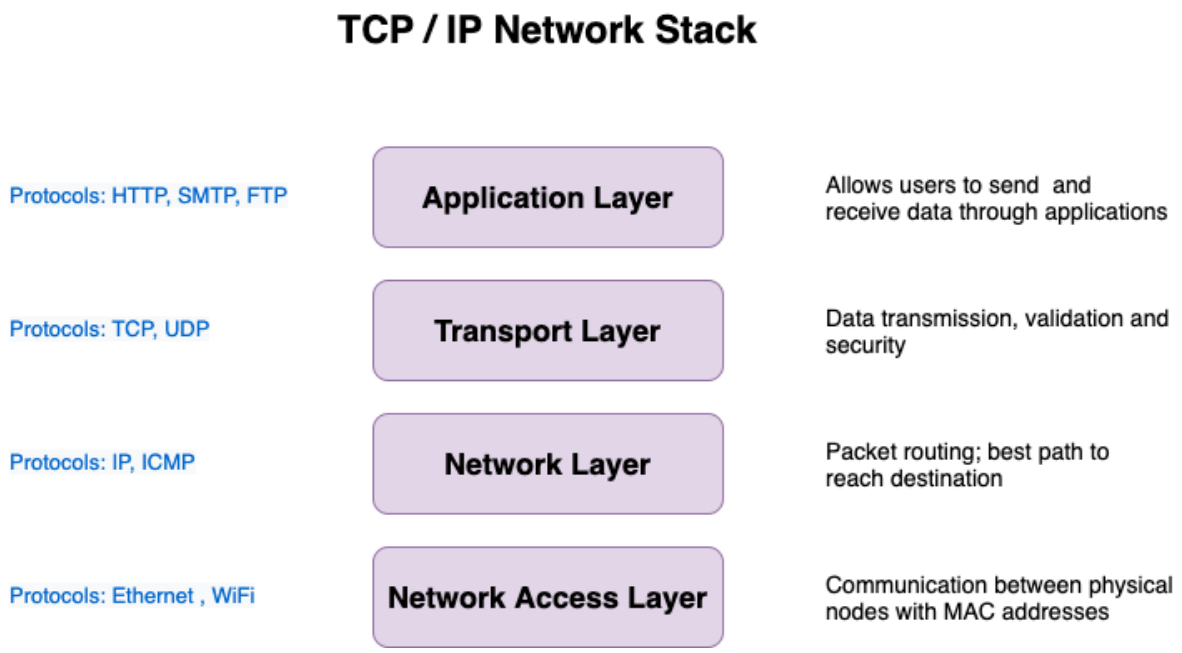

Figure 2.1 TCP/IP network model

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("127.0.0.1:80")
```

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 project are *tcpclient*, *tcpserver*, *http* and *httpserver*.

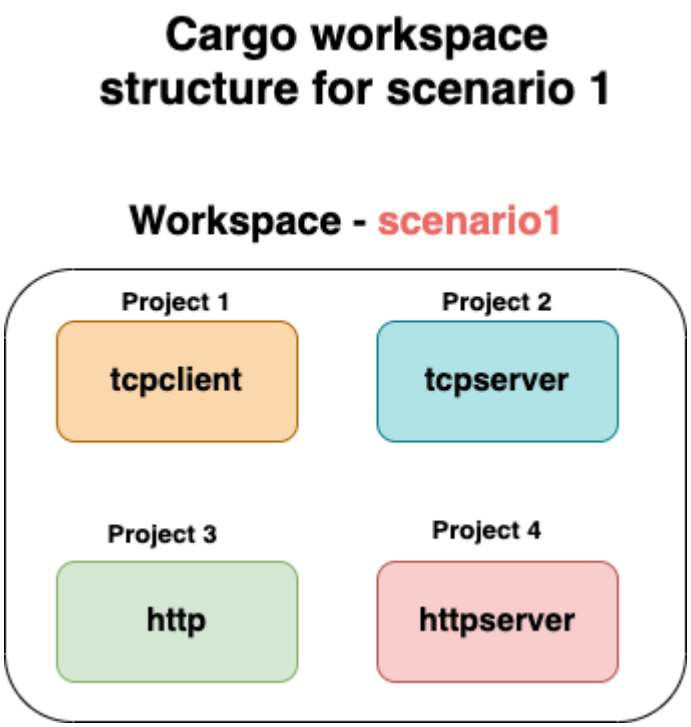

Figure 2.2 Cargo workspace project

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 three 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:

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.

ITERATION 1

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

Listing 2.1 First iteration of TCP server

```
use std::net::TcpListener;

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

❶ Initialize a socket server to bind to IP address 127.0.0.1(localhost) and port 3000

❷ The socket server waits (listens) for incoming connections

❸ When a new connection comes in, it is of type *Result<TcpStream,Error>*, which when unwrapped returns a *TcpStream* if successful, or exits the program with a panic, in case of connection error.

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

```
cargo run -p tcpserver     ❶
```

❶ -p argument specifies which package in workspace we want to run

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.

tcpclient/src/main.rs

```
use std::net::TcpStream;

fn main() {
```

```
    let _stream = TcpStream::connect("localhost:3000").unwrap();          ❶
}
```

❶ The TCP client initiates a connection to remote server running on localhost:3000.

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 2.2 Completing the TCP server

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

❶ TcpStream implements Read and Write traits. So include std::io module to bring Read and Write traits into scope

❷ Make the stream mutable so you can read and write to it.

❸ Read from incoming stream

❹ Echo back whatever is received , to the client on the same connection

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 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*. Implementors 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. Implementors 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 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 2.3 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();    ❶
    let mut buffer = [0; 5];
    stream.read(&mut buffer).unwrap();            ❷
    println!(
        "Got response from server:{:?}",         ❸
        str::from_utf8(&buffer).unwrap()
    );
}
```

❶ Write a "Hello" message to the TCP server connection

❷ Read the bytes received from server

❸ Print out what is received from server. The server sends raw bytes and we have to convert it into UTF-8 str type to print it to terminal.

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() mehod**
>
> 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.

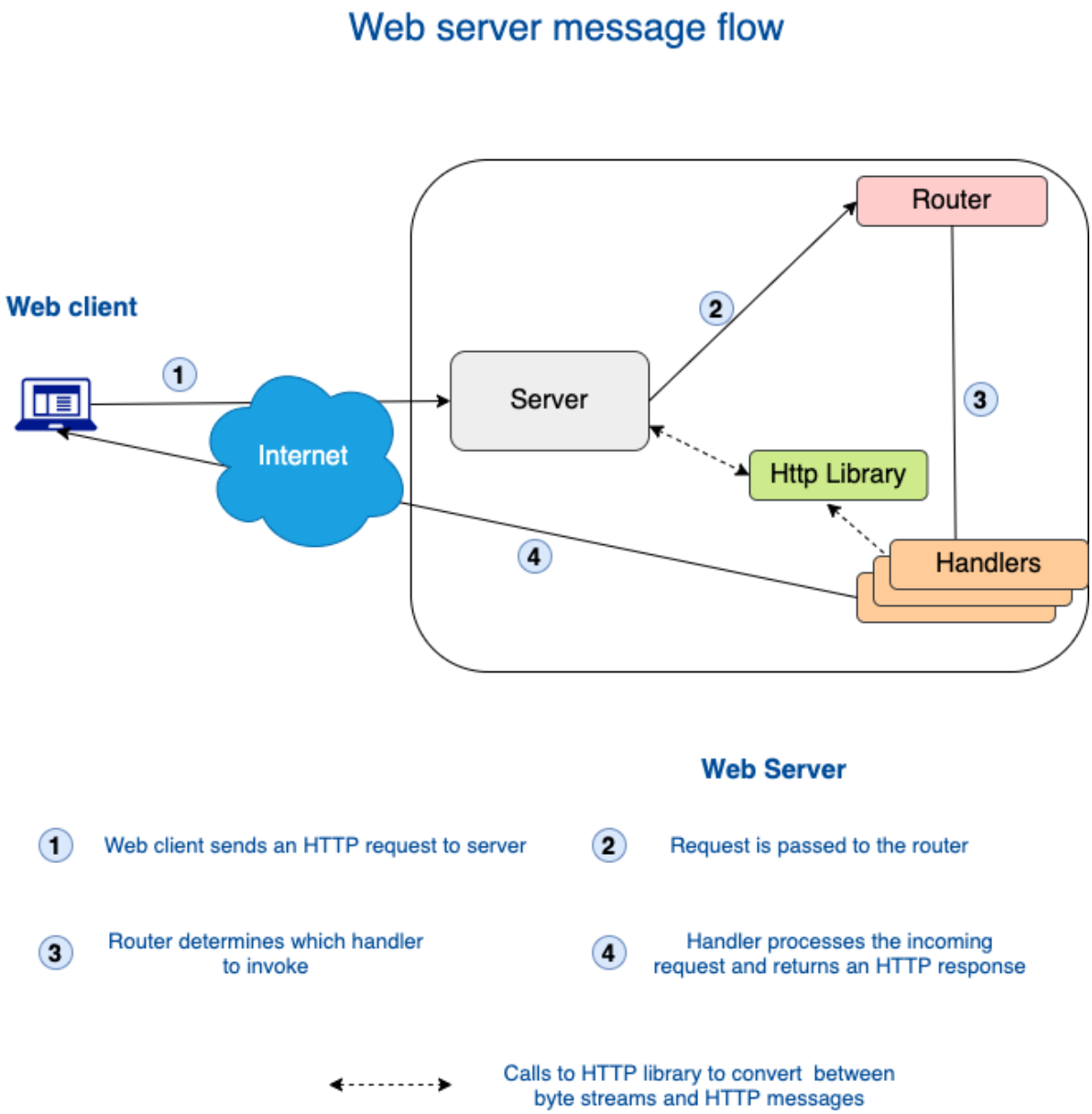

Figure 2.3 Web server message flow

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 byte stream using HTTP library, which is then sent back to 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.

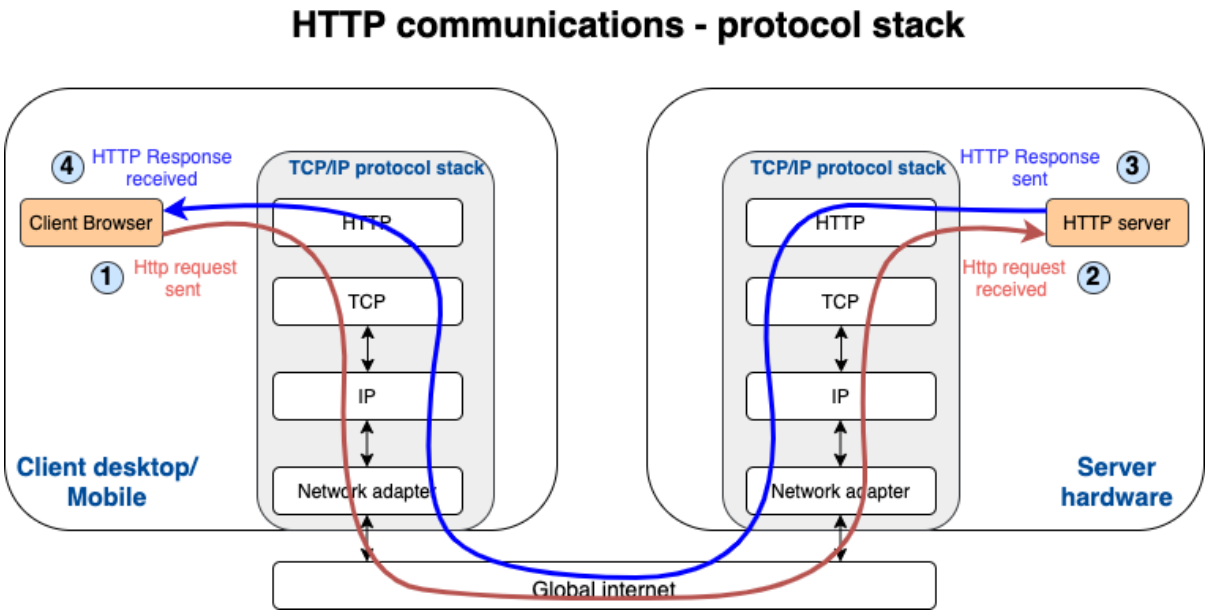

Figure 2.4 HTTP communications with protocol stack

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.

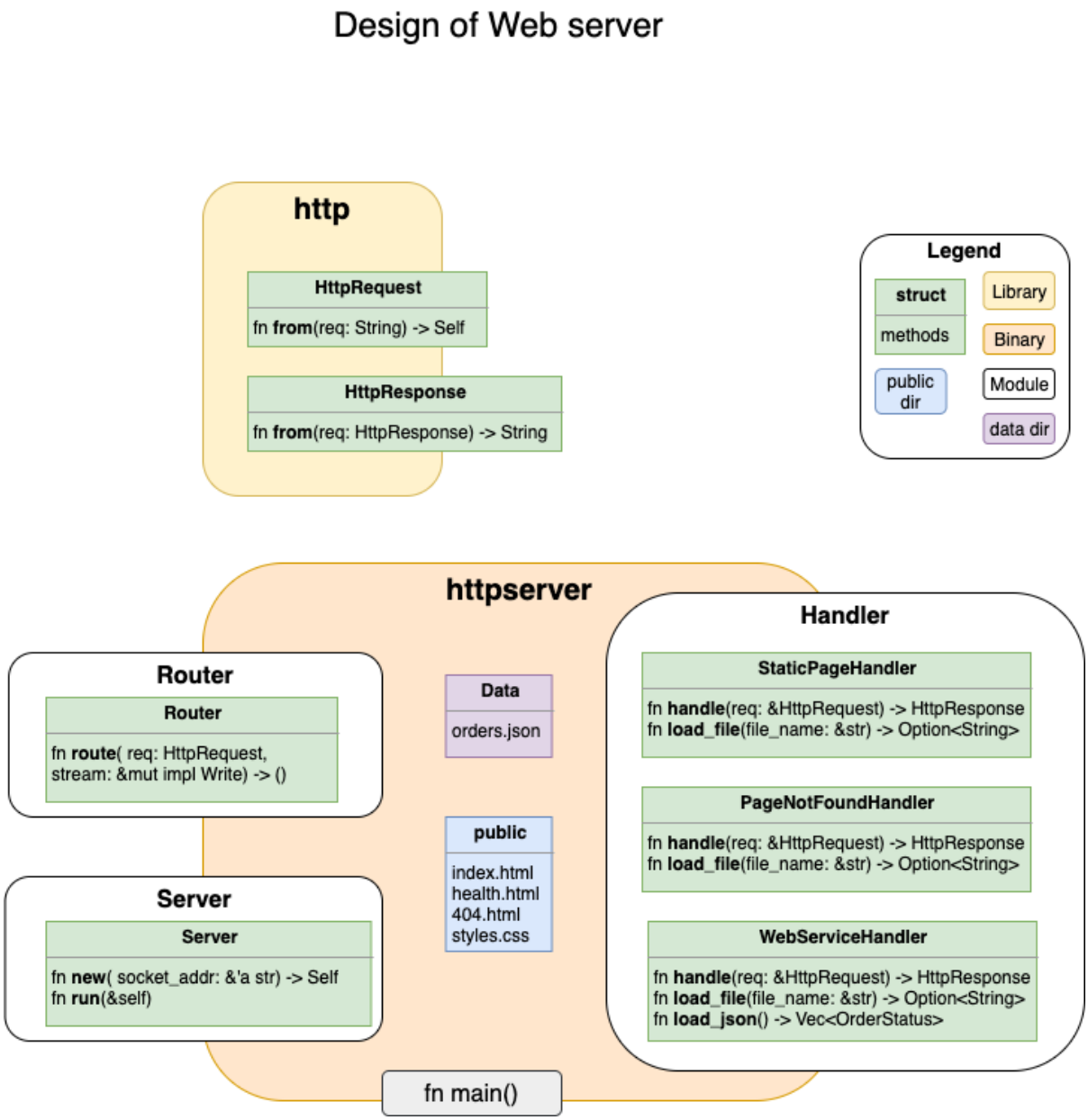

Figure 2.5 Design overview of web server

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.

> **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.

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 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 trait implemented by the data structures for HTTP request.

Rust trait implemented	Description
From<&str>	This trait enables conversion of incoming string slice into HttpRequest 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.js*:

```
#[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 filtered out
```

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.js* 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 filtered out
```

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 2.4 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.

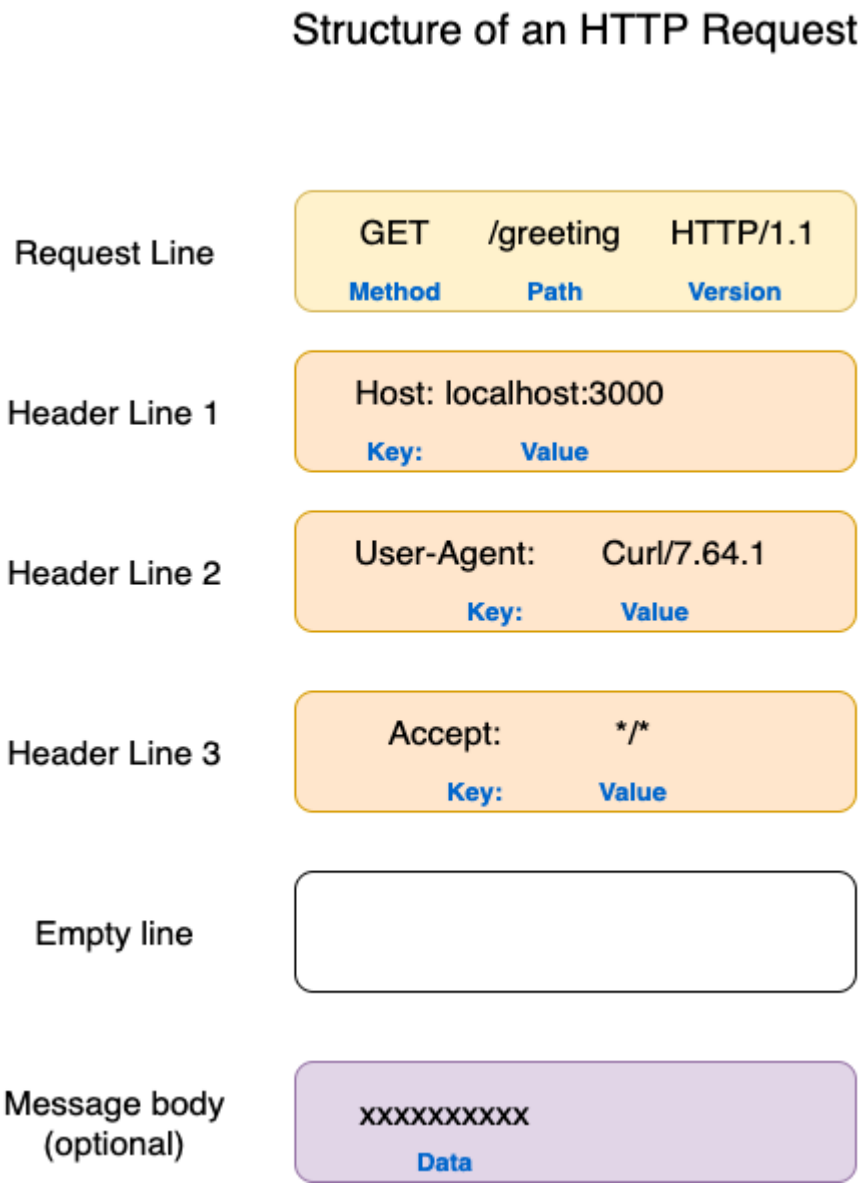

Figure 2.6 Structure of HTTP request

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 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 request line as all request lines contain HTTP keyword and version number), extract the method, path and HTTP version from the line.
    - If line is a header line (identified by separator ':'), extract *key* and *value* for the header

item and add them to 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 line is empty (\n\r), then treat it as separator line. No action is needed in this case
- If 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 2.5 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 incoming HTTP request
        for line in req.lines() {
            // If the line read is request line, call function process_req_line()
            if line.contains("HTTP") {
                let (method, resource, version) = process_req_line(line);
                parsed_method = method;
                parsed_version = version;
                parsed_resource = resource;
            // If the line read is header line, call function process_header_line()
            } 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 2.6 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 whitespaces.
    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 2.7 Parsing incoming HTTP requests: process_header_line() function

```
fn process_header_line(s: &str) -> (String, String) {
    // Parse the headerline 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 2.8 Test scripts for parsing HTTP requests

```
#[test]
fn test_read_http() {
    let s: String = String::from("GET /greeting HTTP/1.1\r\nHost: localhost:3000\r\nUser-Ag
    curl/7.64.1\r\nAccept: */*\r\n\r\n");                ❶
    let mut headers_expected = HashMap::new();    ❷
    headers_expected.insert("Host".into(), " localhost".into());
    headers_expected.insert("Accept".into(), " */*".into());
    headers_expected.insert("User-Agent".into(), " curl/7.64.1".into());
    let req: HttpRequest = s.into();              ❸
    assert_eq!(Method::Get, req.method);          ❹
    assert_eq!(Version::V1_1, req.version);       ❺
    assert_eq!(Resource::Path("/greeting".to_string()), req.resource);   ❻
    assert_eq!(headers_expected, req.headers);                ❼
}
```

❶ Simulate incoming HTTP request

❷ Construct expected headers list

❸ Parse the entire incoming multi-line HTTP request into HttpRequest struct

❹ Verify if method is parsed correctly

❺ Verify if HTTP version is parsed correctly

❻ Verify if the path (resource URI) is parsed correctly

❼ Verify if headers are parsed correctly

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 filtered out
```

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.

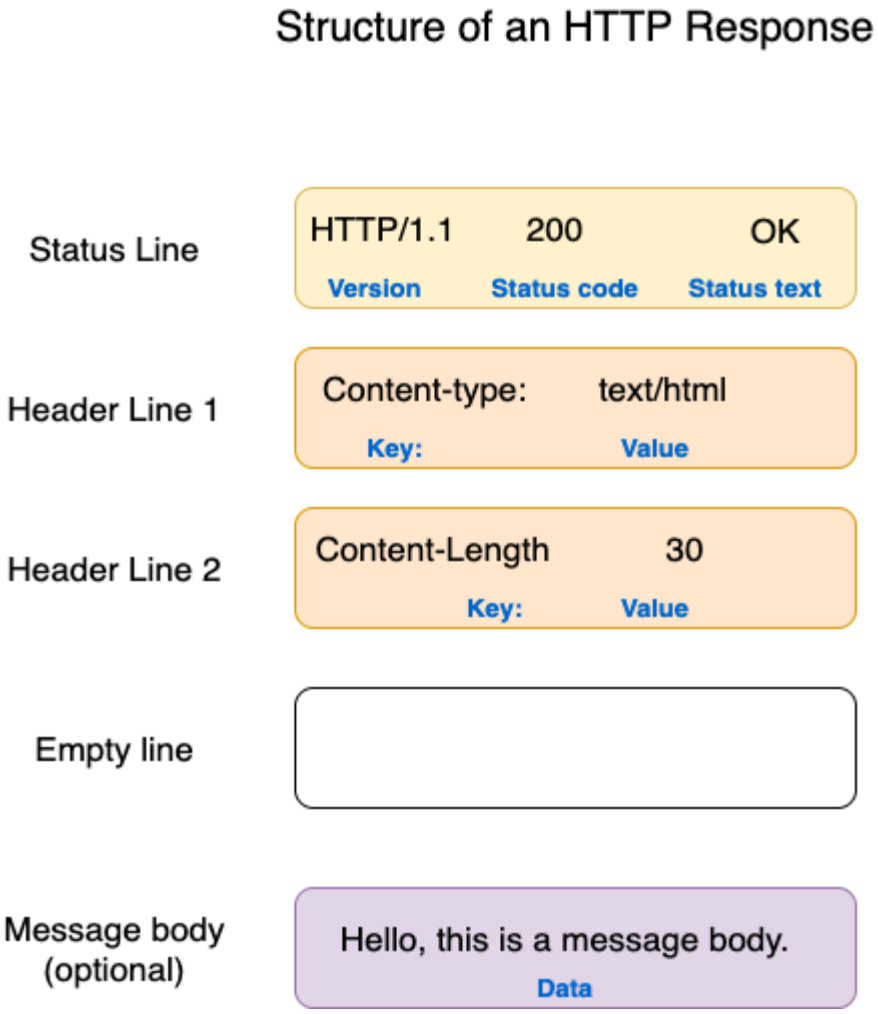

Figure 2.7 Structure of HTTP response

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 2.9 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 inderlying 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 *HttpStruct* 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 2.10 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 2.11 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) -> Result<()> {
        let res = self.clone();
        let response_string: String = String::from(res);
        let _ = write!(write_stream, "{}", response_string);
        Ok(())
    }
}
```

This method accepts a TCP Stream 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 2.12 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().unwrap();
        let mut header_string: String = "".into();
        for (k, v) in map.iter() {
            header_string = format!("{}{}:{}\r\n", header_string, k, v);
        }
        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 2.13 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 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] annotation
}
```

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

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

Listing 2.14 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 2.15 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 right format. Add the following test *within* the *mod tests {}* block,

after the test function *test_response_struct_creation_404()*.

Listing 2.16 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-type:text/html\r\nContent-Leng
    33\r\n\r\nItem was shipped on 21st Dec 2020";
    assert_eq!(http_string, response_actual);
}
```

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

```
cargo test -p http
```

You should see the following message showing that 6 tests have passed in 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 ... ok
test httprequest::tests::test_read_http ... ok
test httpresponse::tests::test_response_struct_creation_404 ... ok

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

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-Length:
    33\r\n\r\nItem was shipped on 21st Dec 2020";
```

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**.

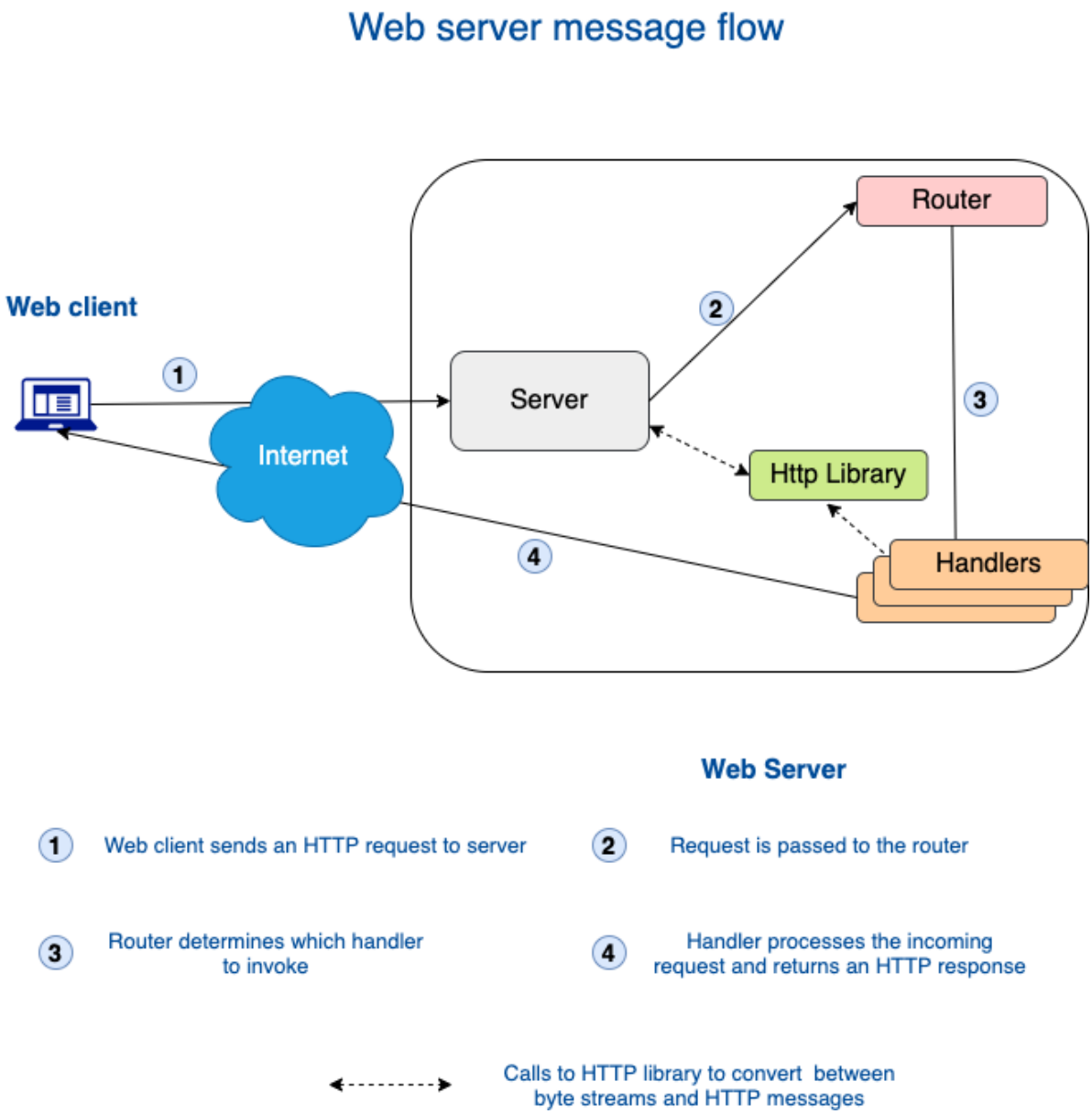

Figure 2.8 Web server message flow

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 2.17 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 server module in *httpserver/src/server.rs*.

Listing 2.18 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).unwrap();
        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.to_vec()).unwrap().into();
            // 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 2.19 Router module

```
use super::handler::{Handler, PageNotFoundHandler, StaticPageHandler, WebServiceHandler};
use http::{httprequest, httprequest::HttpRequest, httpresponse::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 Web service
                        "api" => {
                            let resp: HttpResponse = WebServiceHandler::handle(&req);
                            let _ = resp.send_response(stream);
                        }
                         // Else, invoke static page handler
                        _ => {
                            let resp: HttpResponse = StaticPageHandler::handle(&req);
                            let _ = resp.send_response(stream);
                        }
                    }
                }
            },
            // If method is not GET request, return 404 page
            _ => {
                let resp: HttpResponse = PageNotFoundHandler::handle(&req);
                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::HttpRequest, httpresponse::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 2.20 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_MANIFEST_DIR"));
        let public_path = env::var("PUBLIC_PATH").unwrap_or(default_path);
        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 2.21 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 2.22 Handler to serve static web pages

```
impl Handler for StaticPageHandler {
    fn handle(req: &HttpRequest) -> HttpResponse {
        // Match against the path of static page resource being requested
        match &req.resource {
            http::httprequest::Resource::Path(s) => {
                // Parse the URI
                let route: Vec<&str> = s.split("/").collect();
                match route[1] {
                    "" => HttpResponse::new("200", None, Self::load_file("index.html")),
                    "health" => HttpResponse::new("200", None, Self::load_file("health.html
                    path => match Self::load_file(path) {
                        Some(contents) => {
                            let mut map: HashMap<&str, &str> = HashMap::new();
                            if path.contains(".css") {
                                map.insert("Content-type", "text/css");
                            } else if path.contains(".js") {
                                map.insert("Content-type", "text/javascript");
                            } else {
                                map.insert("Content-type", "text/html");
                            }
                            HttpResponse::new("200", Some(map), Some(contents))
                        }
                        None => HttpResponse::new("404", None, Self::load_file("404.html"))
                    },
                }
            }
        }
    }
}
```

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 consructed
- 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 file is not found, it sends back a 404 error page. If 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 2.23 Handler to serve json data

```
// Define a load_json() method to load orders.json file from disk
impl WebServiceHandler {
    fn load_json() -> Vec<OrderStatus> {
        let default_path = format!("{}/data", env!("CARGO_MANIFEST_DIR"));
        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()).unwrap();
        orders
    }
}
// Implement the Handler trait
impl Handler for WebServiceHandler {
    fn handle(req: &HttpRequest) -> HttpResponse {
        match &req.resource {
            http::httprequest::Resource::Path(s) => {
                // 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()).unwrap())
                        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("404.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.

Listing 2.24 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;
}
```

Listing 2.25 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>
```

Listing 2.26 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 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 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://github.com/peshwar9/rust-servers-services-apps.

2.3 Summary

- The TCP/IP model is a simplified set of standards and protocols for communication over the internet. It is organied 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://github.com/peshwar9/rust-servers-services-apps.

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 = "3.1.0"             ❶
actix-rt = "1.1.1"              ❷
```

❶ You can use this version of actix-web or whichever later version is available at the time you are reading this.

❷ Async run-time for Actix. Rust requires use of external run-time engine for executing async code.

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

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

// Instantiate and run the HTTP server
#[actix_rt::main]
async fn main() -> io::Result<()> {
    // Construct app and configure routes  ❸
    let app = move || App::new().configure(general_routes);

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

❶ For HTTP GET requests coming in on route /health, the Actix web server will route the request to health_check_handler()

❷ The handler constructs an HTTP Response with a greeting

❸ Construct an actix web application instance and register the configured routes.

❹ Initialize a web server, load the application, bind it to a socket and run the server.

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.

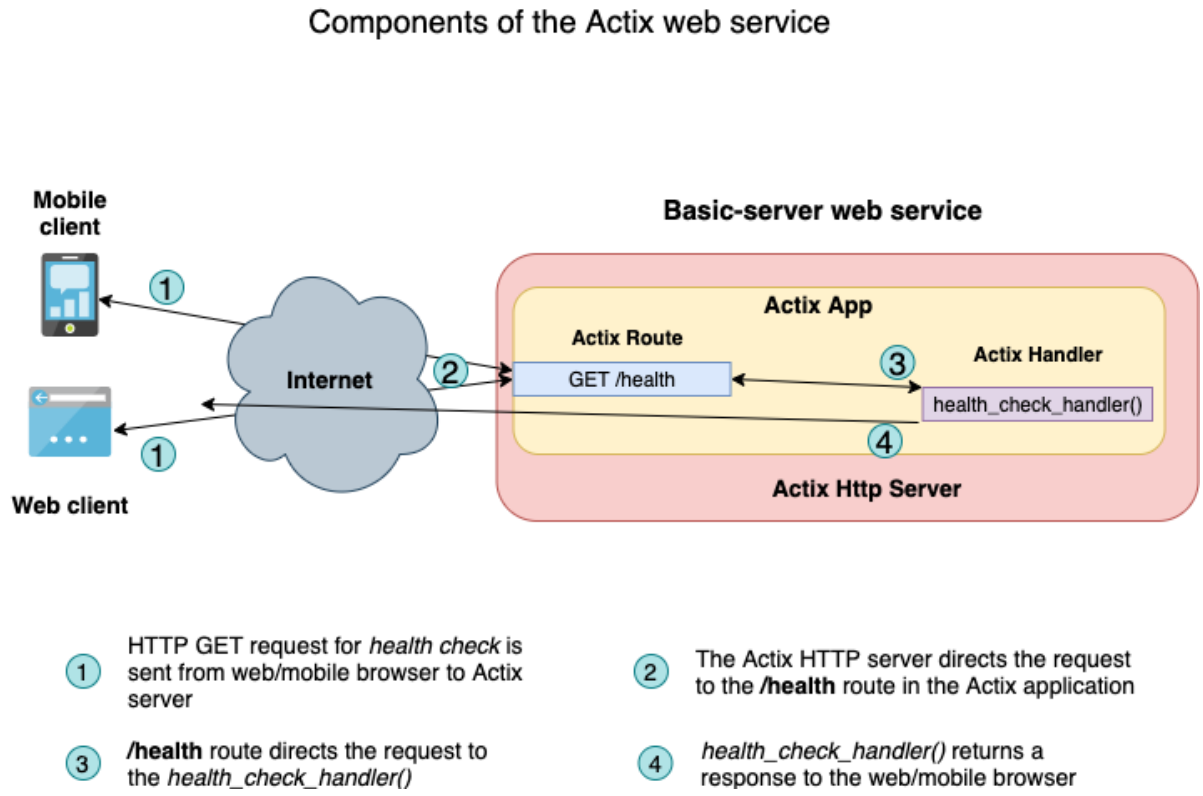

Figure 3.1 Actix Basic server

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 the 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.

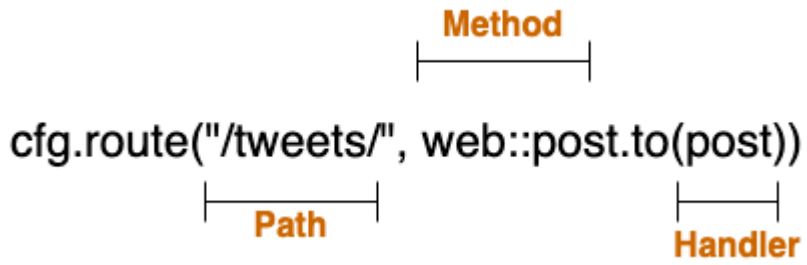

Figure 3.2 Structure of Actix route

This is the route we implemented earlier for health check:

```
cfg.route(
    "/health",                      ❶
    web::get()                      ❷
    .to(health_check_handler));     ❸
```

❶ Path

❷ HTTP method

❸ Request handler method

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 kicking")
}
```

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/../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/..berlin 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.

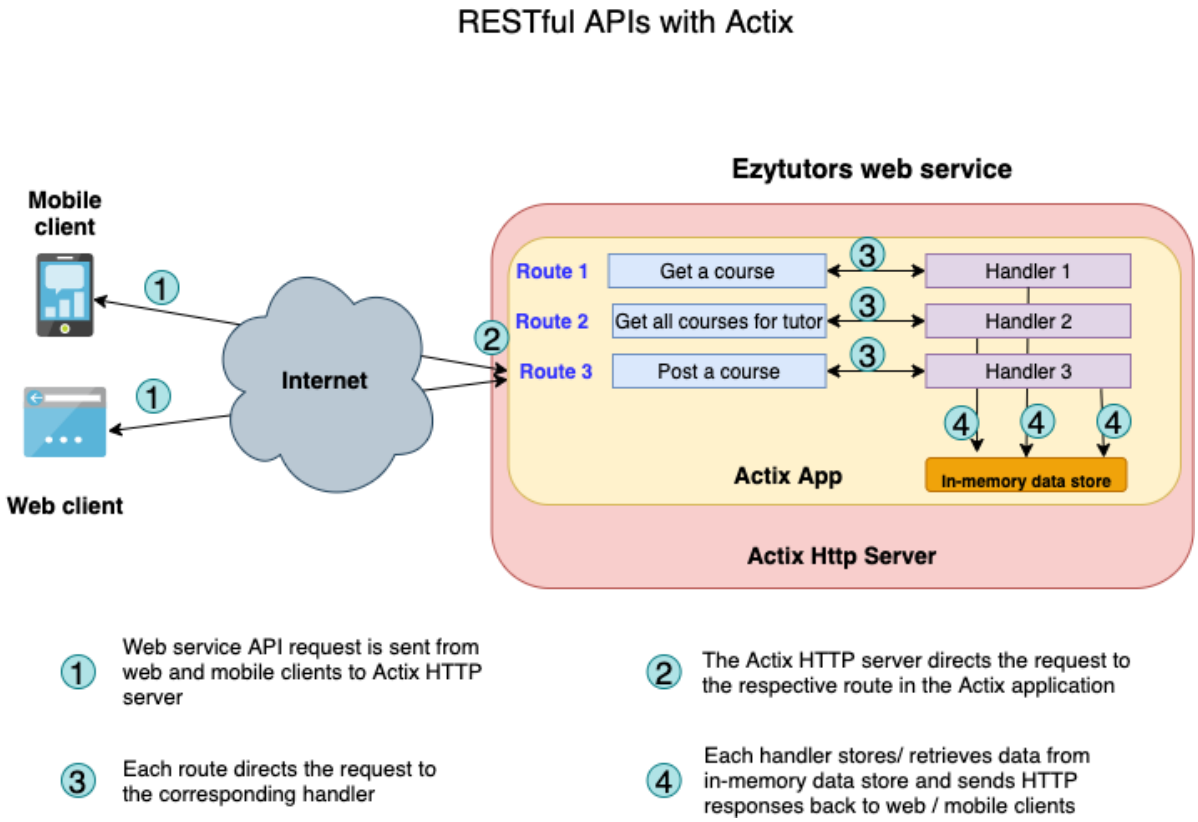

Figure 3.3 Components of the web service

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.

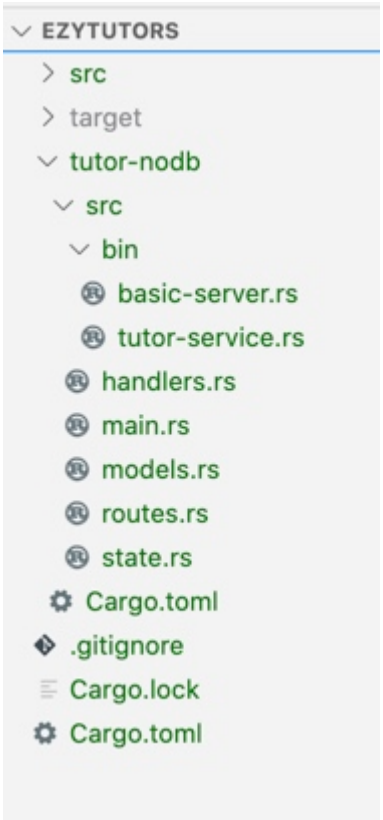

Figure 3.4 Project structure of EzyTutors web service

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 wll 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 build 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. Ofcourse, 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/../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 deling 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 framweork 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,        ❶
    pub visit_count: Mutex<u32>,              ❷
}
```

❶ Shared immutable state

❷ Shared mutable state

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 ".to_string(),
        visit_count: Mutex::new(0),
    });
    let app = move || {                                            (2)
        App::new()
            .app_data(shared_data.clone())                         (3)
            .configure(general_routes)                             (4)
    };

    HttpServer::new(app).bind("127.0.0.1:3000")?.run().await       (5)
}
```

(1) Initialize application state

(2) Define the web application

(3) Register application state with the web application

(4) Configure routes for the web application

(5) Initialize Actix web server with the web application, listen on port 3000 and run the server

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>) -> HttpResponse {
  ❶
    let health_check_response = &app_state.health_check_response;     ❷
    let mut visit_count = app_state.visit_count.lock().unwrap();      ❸
    let response = format!("{} {} times", health_check_response, visit_count);    ❹
    *visit_count += 1;                ❺
    HttpResponse::Ok().json(&response)
}
```

❶ Application state registered with the Actix web application is made available automatically to all handler functions as an extractor object of type web::Data<T> where T is the type of the custom application state that developers have defined.

❷ Data members of the Application state struct (*AppState*) can be directly accessed using standard dot notation

❸ Field representing shared mutable state (*visit_count*) has to be locked first before accessing, to prevent multiple threads from updating the value of the field simultaneously

❹ Construct response string to send back to browser client

❺ Update value of the field representing shared mutable state. Since the lock on this data has already been acquired, the value of the field can be updated safely. The lock on the data is automatically released when the handler function finishes execution.

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)]   ❶
pub struct Course {
    pub tutor_id: usize,
    pub course_id: Option<usize>,
    pub course_name: String,
    pub posted_time: Option<NaiveDateTime>,          ❷
}
impl From<web::Json<Course>> for Course {              ❸
    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,
        }
    }
}
```

❶ The annotation #derive derives the implementations for four traits - Deserialize, Serialize, Debug and Clone. The first two are part of the Serde crate and help to convert Rust data structs into on-the-wire formats and vice versa. Implementing Debug trait helps to print the Course struct values for debug purposes. Clone trait helps address the Rust ownership rules during processing.

❷ NativeDateTime is a chrono data type for storing timestamp information

❸ Function to convert data from incoming HTTP request to Rust struct

In the code shown, you will notice that *course_id* and *posted_time* have been declared to be of type Option<usize> and Option<NaiveDateTime> respectively. What this means is that these

two fields can either hold a valid value of type *usize* 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 TrainingInstite;
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
    offering for business customers
}
```

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>>,          ❶
}
```

❶ Courses are stored in application state as a Vec collection, protected by a Mutex.

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 ".to_string(),
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),          ❶
    });
// other code
}
```

❶ courses field initialized with a Mutex-protected empty vector

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 1
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/json"
    -d '{"tutor_id":1, "course_name":"Hello , my first course  !"}'
```

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)          ❶
};
```

❶ Register the new course_routes group with application

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)   (4)
        .collect::<Vec<Course>>()                           (5)
        .len();                                             (6)
    let new_course = Course {                               (7)
        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);     (8)
    HttpResponse::Ok().json("Added course")                 (9)
}
```

(1) The handler function takes two parameters - data payload from HTTP request and application state.

(2) Since courses collection is protected by Mutex, we have to lock it first to access the data

(3) Convert the course collection (stored within AppState) into an iterator, so that we can iterate through each element in the collection for processing

(4) Review each element in collection and filter only for the courses corresponding to the tutor_id (received as part of the HTTP request)

(5) The filtered list of courses for the tutor is stored in a Vector

(6) The number of elements in filtered list is computed. This is used to generate the id for next course.

❼ Create a new course instance

❽ Add the new course instance to the course collection that is part of the application state (*AppState*)

❾ Send back an HTTP response to web client

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)]                                          ❶
mod tests {                                           ❷
    use super::*;                                     ❸
    use actix_web::http::StatusCode;
    use std::sync::Mutex;

    #[actix_rt::test]                                 ❹
    async fn post_course_test() {
        let course  = web::Json(Course {              ❺
            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(AppState {        ❻
            health_check_response: "".to_string(),
            visit_count: Mutex::new(0),
            courses: Mutex::new(vec![]),
        });
        let resp = new_course(course, app_state).await;                       ❼
        assert_eq!(resp.status(), StatusCode::OK);                            ❽
    }
}
```

❶ The #[cfg(test)] annotation on tests module tells Rust to compile and run the tests only when cargo test command is run, and not for cargo build or cargo run

❷ Tests in Rust are written within the tests module.

❸ Import all handler declarations from the parent module (which hosts the tests module)

❹ Normal rust tests are annotated with #[tests]. But since this is an asynchronous test function, we have to alert the async run-time of Actix-web to execute this async test function.

❺ Construct a web::Json<T> object representing request data payload , i.e. new course data from tutor

❻ Construct a web::Data<T> object representing application state

❼ Invoke handler function with application state and simulated request data payload

❽ Verify if the HTTP status response code (returned from the handler) indicates success

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 filtered out
```

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"
    -d '{"tutor_id":1, "course_name":"Hello , my first course  !"}'
```

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("/{user_id}", web::get().to(get_courses_for_tutor)),   ❶
    );
}
```

❶ Add new route for getting courses for a tutor (represented by the user_id variable)

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<(usize)>,
) -> HttpResponse {
    let tutor_id: usize = params.0;

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

    if filtered_courses.len() > 0 {
        HttpResponse::Ok().json(filtered_courses)         ❷
    } else {
        HttpResponse::Ok().json("No courses found for tutor".to_string())    ❸
    }
}
```

❶ Filter for courses corresponding to tutor requested by web client

❷ If courses are found for tutor, return success response with the course list

❸ If courses are not found for tutor, send error message.

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:3000/1
```

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

```
.route("/{user_id}", web::get().to(get_courses_for_user))
```

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(AppState {    ❶
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),
    });
    let tutor_id: web::Path<(usize)> = web::Path::from((1));          ❷
    let resp = get_courses_for_tutor(app_state, tutor_id).await;          ❸
    assert_eq!(resp.status(), StatusCode::OK);                        ❹
}
```

❶ Construct app state

❷ Simulate request parameter

❸ Invoke the handler

❹ Check response

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" -d
    '{"tutor_id":1, "course_name":"Hello , my first course  !"}'
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json" -d
'{"tutor_id":1, "course_name":"Hello , my second course  !"}'
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json" -d
'{"tutor_id":1, "course_name":"Hello , my third course  !"}'
```

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  !"
    ,"posted_time":"2020-09-05T06:26:51.866230"},
    {"tutor_id":1,"course_id":2,"course_name":"Hello , my second course  !"
    ,"posted_time":"2020-09-05T06:27:22.284195"},{"tutor_id":1,"course_id":3,"course_name"
    :"Hello , my third course  !" ,"posted_time":"2020-09-05T06:57:03.850014"}]
```

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("/{user_id}", web::get().to(get_courses_for_user))
            .route("/{user_id}/{course_id}", web::get().to(get_course_detail)),     ❶
    );
}
```

❶ Add new route 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<(usize, usize)>,
) -> 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(course_id))   ❶
        .ok_or("Course not found");                                           ❷

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

❶ Retrieve course corresponding to the tutor_id and course_id sent as request parameters.

❷ Converts Option<T> to Result<T,E>. If Option<T> evaluates to Some(val), it returns Ok(val). If None found, it returns Err(err).

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:3000/1/1
```

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

```
.route("/{user_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(AppState {    ❶
        health_check_response: "".to_string(),
        visit_count: Mutex::new(0),
        courses: Mutex::new(vec![]),
    });
    let params: web::Path<(usize, usize)> = web::Path::from((1, 1));      ❷
    let resp = get_course_detail(app_state, params).await;                ❸
    assert_eq!(resp.status(), StatusCode::OK);                            ❹
}
```

❶ Construct app state

❷ Construct an object of type web::Path with two parameters. This is to simulate a user typing localhost:3000/1/1 in a web browser.

❸ Invoke the handler

❹ Check response

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"
    -d '{"tutor_id":1, "course_name":"Hello , my first course  !"}'
curl -X POST localhost:3000/courses/ -H "Content-Type: application/json"
-d '{"tutor_id":1, "course_name":"Hello , my second course  !"}'
```

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  !"
    ,"posted_time":"2020-09-05T06:26:51.866230"}
```

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.