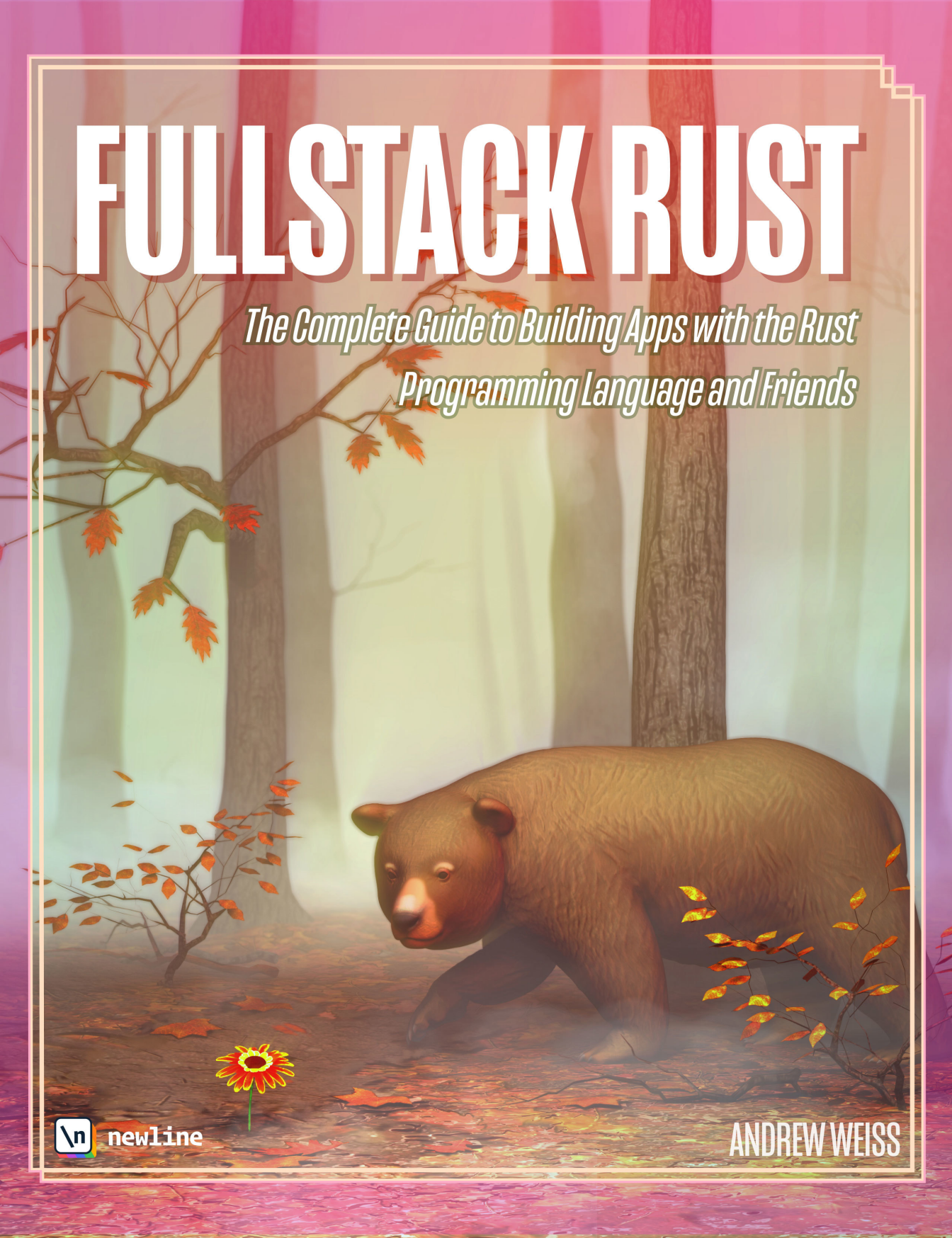
FULLSTACK RUST

The Complete Guide to Building Apps with the Rust Programming Language and Friends



newline

ANDREW WEISS

Fullstack Rust

The Complete Guide to Buildings Apps with the Rust Programming Language and Friends

Written by Andrew Weiss
Edited by Nate Murray

Published by Fullstack.io.

Contents

Book Revision

Revision 5 - 2020-02-20

Join Our Discord

Come chat with other readers of the book in the official newline Discord channel:

Join here: https://newline.co/discord/rust[1]

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[2].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[3].

[1] https://newline.co/discord/rust
[2] https://twitter.com/fullstackio
[3] mailto:us@fullstack.io

Introduction

There are numerous reasons to be hopeful about the future of computing, one of which is the existence and continued progression of the Rust programming language.

We are currently in the fifth era of programming language evolution. This is an era where languages have been able to take all of the learnings since the 1950s and incorporate the best parts into languages each with its own cohesive vision.

We have specialized languages cropping up for a wide variety of tasks and countless general purpose languages being actively developed and used. There are significant resources in industry to invest in language design and development which compliment the vibrant academic community. With tools like LLVM and the explosion of open source, creating a language has never been easier.

It is in this environment that Rust has been voted the "most loved programming language" in the Stack Overflow Developer Survey every year since 2016. Standing out in this increasingly crowded world of languages is enough of a reason to ask why Rust?

Why Rust?

There are a few potential readings of this question: why should I learn Rust, why are others using Rust, why should I choose Rust over language X? These are all relevant, but I want to start with a bit of a philosophical argument for Rust independent of these specific points.

There is a limit to how transformative an experience you can have when learning a language in a similar paradigm to one you already know. Every language and paradigm has an intrinsic style that is forced on you as you try to solve problems.

If you work within that style then your code will flow naturally and the language will feel like it is working with you. On the other hand, if you fight the natural style of the language you will find it hard or impossible to express your ideas.

Moreover, learning and working with a language will teach you ways to be more effective based on how the language guides you based on its natural design. How much you are able to learn is a function of how much your prior experience and mental models cover the new language.

Rust borrows a lot of ideas from other languages and is truly multi-paradigm, meaning you can write mostly functional code or mostly imperative code and still fit nicely within the language. The most unique feature of the language, the borrow checker, is a system that enforces certain invariants which allow you to make certain safety guarantees. Even this is built on prior art found in earlier languages.

All of these good ideas from the world of programming language design combine in a unique way to make Rust a language that truly makes you think about writing code from a novel perspective. It does not matter how much experience you have, learning Rust will forever change the way you write code for the better.

Okay with that philosophical argument out of the way, let's dig in to some specifics of why Rust is a exciting.

To help guide this discussion, we can break things down into a few broad categories.

On language comparisons

There is no best programming language. Almost every task has a variety of languages which could be the right tool. Every language comes with good parts and bad parts. Evaluating these trade-offs when faced with a particular problem space is an art unto itself. Therefore, nothing in this book is intended to disparage or denigrate any particular alternative language. The primary goal of this book is to faithfully present Rust. That being said, sometimes comparisons with other languages are instructive and are meant to be instructive rather than as fuel in a flame war.

Language features

There are a lot of features of Rust which make it a great tool for a great number of tasks. Some highlights include:

- Performance

- Strong, static, expressive type system
- Great error messages
- Modern generics
- Memory safety
- Fearless concurrency
- Cross platform
- C interoperability

Let's briefly go through some of these which are probably the biggest reasons that Rust gets talked about.

Performance

Rust is exceptionally fast, in the same ballpark as C and C++. For some programs, specifically due to the lack of pointer aliasing, the Rust compiler can sometimes have enough information to optimize code to be faster than what is possible in C without directly writing assembly. For the vast majority of use cases, you should consider Rust to be fast enough.

Often the most obvious way to write a program is also the fastest. Part of this comes from the commitment to zero-cost abstractions, which are summarized by Bjarne Stroustrup, the creator of C++, as:

> What you don't use, you don't pay for. And further: What you do use, you couldn't hand code any better.

Most of the abstractions in Rust, for example iterators, are zero-cost by this definition. The most efficient way to traverse a vector of data is to use a for loop which uses an iterator trait. The generated assembly is usually as good as you could hope for had you written it by hand.

The other aspect of performance is memory consumption. Rust does not have a garbage collector so you can use exactly as much memory as is strictly necessary at any given time. Due to the design of the language, you start to think and see every memory allocation. Using less memory is often easier than the converse. The rest of the language is designed around making working without a garbage collector painless.

Type system

The type system of Rust is influenced by the long lineage of functional programming languages such as ML and Haskell. It is static, nominal, strong, and for the most part inferred. Don't worry if that didn't mean anything to you, but if it did then great. You encode the ideas and constraints of your problem with types. You only have to specify types in a few places with the rest able to be inferred. The compiler then checks everything for you so that you get faster feedback about potential problems. Entire classes of bugs are impossible because of static typing. Most things you encounter in practice are expressible in the type system. The compiler then checks everything for you so that you get faster feedback about potential problems. Entire classes of bugs are impossible because of static typing.

A type system is often called expressive if it is easy to encode your ideas. There are some concepts which are impossible to express in static type systems. Rust has powerful abstraction facilities like sum and product types, tuples, generics, etc. which put the type system definitely in the expressive camp.

Memory safety

A language is memory safe if certain classes of bugs related to memory access are not possible. A language can be called memory unsafe if certain bugs are possible. A non-exhaustive list of memory related bugs include: dereferencing null pointers, use-after free, dangling pointers, buffer overflows.

If you have never written code in a memory unsafe language then these might sound like gibberish to you, which is fine. The important point is this class of bugs is a consistent and large source of security vulnerabilities in systems implemented with memory unsafe languages. For example, about 20% of CVEs[4] ever filed against the Linux kernel are due to memory corruption or overflows. Linux is implemented primarily in C, a spectacularly memory unsafe language.

Memory safety bugs are bad for security and reliability. They lead to vulnerabilities and they lead to crashes. If you can rule these out at compile time then you are in a much better state of the world.

Rust is designed to be memory safe, and thus it does not permit null pointers, dangling pointers, or data races in safe code. There are many interacting features

[4] https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33

which allow this guarantee. The primary one is the unique system of ownership combined with the borrow checker. This is part of the compiler that ensures pieces of data live at least as long as they need to in order to be alive when they are used.

One other feature is the builtin `Option` type. This is used to replace the concept of null found in many other languages. In some languages, every type is secretly the union of that type with null. This means that you can always end up with bugs where you assume some variable had a value and it actually was inhabited by the dreaded null. Rust disallows this by not having null and instead having a type which can explicitly wrap other types. For example, consider this Rust code:

```
fn print_number(num: Option<i32>) {
  match num {
    Some(n) => println!("I see {}!", n),
    None => println!("I see nothing!"),
  }
}

fn main() {
  let x = Some(42);
  let y = None;

  print_number(x);
  print_number(y);
}
```

The function `print_number` must handle the case where `num` is `None`, meaning the `Option` has no value. There are a few different ways to handle that case but you must explicitly do something for that case or else your code will not compile.

The one caveat here is that Rust does allow blocks of code to be marked `unsafe` and within those blocks it is possible to violate memory safety. Some things are impossible for the compiler to verify are safe and therefore it refuses to do so. It requires you to use `unsafe` regions of code to ensure that you understand the invariants required to make sure your code truly is safe.

This does not defeat the purpose, rather in isolates the areas of auditability to just those sections of code which are specifically marked. Nothing you do in normal Rust,

also called safe Rust, can result in a memory safety violation, unless something in unsafe code did something wrong ahead of you.

As an example, calling C functions from Rust is unsafe. This is because Rust has no way of knowing what the C code is doing, and C is inherently unsafe, therefore the compiler cannot uphold its guarantees if you call out to C. However, can it be safe to call C? Yes, provided you fill in the visibility gap for the compiler with your own logic.

Fearless concurrency

Concurrency in programming means that multiple tasks can be worked on at the same time. This is possible even for a single thread of execution by interleaving the work for different tasks in chunks rather than only working on tasks as entire chunks.

Parallelism in programming means multiple tasks executing at the exact same time. True parallelism requires multiple threads of execution (or the equivalent).

The Rust language describes its facilities for concurrent and parallel computing as fearless concurrency with a bit of conflation of terms. I will continue in this tradition and use concurrency to mean concurrency and/or parallelism.

Most modern, high level languages have chosen how they want to support concurrency and mostly force you down that path. Some more general purpose languages provide the tools to handle concurrency however you see fit. For example, Go is designed around Communicating Sequential Processes (CSP) and therefore concurrency is most easily achieved using channels and goroutines. Python, on the other hand, has libraries for threads, multiprocesses, message passing actors, etc.

Rust is a low-level language by design and therefore provides tools that allow you to use the model of your choice to achieve your particular goals. Therefore, there are facilities for threads but also channels and message passing.

Regardless of what technique you choose to tackle concurrency and/or parallelism, the same ownership model and type system that ensures memory safety also ensures thread safety. This means that it is a compile time error to write to the same memory from different threads without some form of synchronization. The details are less important than the concept that entire classes of problems that are notoriously difficult to debug in other languages are completely eliminated at compile time while, importantly, retaining all of the performance benefits.

C interoperability

Rust is foremost a systems programming language. That means it is designed for building low level systems with strict performance requirements and reliability constraints. Frequently in this world, C is the glue that binds many disparate systems. Therefore being able to interoperate with C is an absolute necessity to be able to have a serious systems language. Luckily it is straightforward to interact with C both by calling into C from Rust, as well as exposing Rust as a C library.

You might be saying that sounds great but I don't plan on writing an operating system anytime soon so why should I care? C is also the most common mechanism for making dynamic languages faster. Typically, when parts of your Python or Ruby code are showing performance problems, you can reach for an extension written in C to speed things up. Well, now you can write that extension in Rust and get all of the high level benefits of Rust and still make your Python or Ruby code think it is talking to C. This is also quite an interesting area for interacting with the JVM.

Ecosystem

Software is not constructed in a vacuum, the practice of programming is often a community driven endeavor. Every language has a community whether it actively cultivates it or not. The ecosystem around a language includes the community of people, but also the tooling or lack thereof.

Rust has grown quite a lot in its short life and has gone through some growing pains as a result. However, the community has always been very welcoming and importantly the culture is a first-class citizen. Rust specifically has a community team as part of the governance structure of the language. This goes a long way to helping the language and ecosystem grow and mature.

We will cover much of the useful tooling that exists around Rust in detail below. However, suffice it to say that the tooling around the language is some of the best that exists. There have been a lot of learnings over the past twenty years about how to manage toolchains and dependencies and Rust has incorporated all of this quite well.

The nature of programming

The systems and applications we are building today are different than 50 years ago, they are even different than 10 years ago. Therefore, it should not be too much of a stretch to say that the tools we use should also be different.

There is an explosion of embedded systems due to what is commonly called the Internet of Things. However, is C still the best tool for that job? Mission critical software that controls real objects that could lead to serious consequences in the case of failure should be using the best tool for the job. Rust is a serious contender in this space. For example, it is easy to turn off dynamic memory allocation while still being able to use a lot of the nice parts of the language.

The other explosion is continuing on the web. We have been in a web revolution for quite a while now, but things have not slowed down. The deficits of JavaScript are well known and have been addressed along quite a few paths. We have many languages which compile to JavaScript but provide nice features like type systems or a functional paradigm. However, there are fundamental performance and security issues with JavaScript regardless of how you generate it. WebAssembly (WASM) is a step in a different direction where we can compile languages like Rust to a format natively executable in the browser.

Fun

Rust is fun to write. You will disagree with this and think I am crazy at some point while you are learning Rust. There is a learning curve which can be distinctly not fun. However, once your mental model starts to shift, you will find yourself having moments of pure joy when your code just works after the compiler gives you the okay.

Why not Rust

Rust is just another programming language and as such is just another software project. This means it has built up some legacy, it has some hairy parts, and it has some future plans which may or may not ever happen. Some of this means that for any given project, Rust might not be the right tool for the job.

One area in which Rust might not be right is when interfacing with large C++ codebases. It is possible to have C++ talk to C and then have C talk to Rust and vice versa. That is the approach you should take today if possible. However, Rust does not have a stable ABI nor a stable memory model. Hence, it is not directly compatible with C++. You can incrementally replace parts of a system with Rust and you can build new parts in Rust, but plug-and-play interoperability with C++ is not a solved problem.

Furthermore, Rust takes time to learn. Now this is often cited as a reason for sticking with some other language because one is deemed an expert in that language. However, a counter point might be that you are not as much of an expert in that language as you might believe. A further counter point is that it might not matter, the other language might be fundamentally flawed enough that being an expert is irrelevant. Nonetheless, there are times where using the tool you know is the right answer.

The gap between learning Rust and knowing it from using it in anger is a bit bigger than in some other languages. Therefore the learning curve might seem steeper than you are used to. However, this is primarily because what is safe in Rust with the borrow checker helping you can be insane in other languages.

Type systems are amazing. You tell the computer some facts about your problem domain and it continually checks that those things are true and lets you know if you screw up. Yet there are valid programs which are inexpressible in a static type system. This is both theoretically true and actually happens in practice. Moreover, dynamic languages can frequently be more productive for small, isolated tasks. Sometimes the cost of the type system is not worth it.

This book's mission

Rust has a great set of documentation around the standard library and has an official "book"[5] which is a great place to start if you are looking for another source of material. However, this book has a different focus than a traditional book trying to teach you a language. Our goal is to build realistic applications and explore some of the techniques and tools available in Rust for accomplishing those tasks.

[5] https://doc.rust-lang.org/book/

In the process of working through some common scenarios, hopefully you will also be able to learn Rust. There is a gradual ramp up from very simple to more complex programs as we build up our Rust toolbelt. One specific goal is to show places where many people usually stumble and try to support you in finding your own ways over those hurdles. This should empower you when you branch out to your own problems.

This approach has the downside of not necessarily covering every language feature in the same depth or in the same order that you might encounter in a standard programming language introduction. Furthermore, we will explicitly try to take a pragmatic path rather than belabor esoteric details. Those details can be quite interesting and will be there for you when you want to seek them out, but often they get in the way of learning. We will sometimes do things in a less than perfect way as the trade-off is worth the expositional benefit.

Overall the goal is to get you to a state of productivity as quickly as possible. Along the way we will provide pointers to further material if you want to go deeper.

Setting expectations based on your background

A great deal of terminology in the programming language space is built on a base set of shared ideas that have become so entrenched as to be overlooked by most every day developers. There are some lines we have to draw where we lean on some assumed prior knowledge. Thus, if you have never written any code before this might be a challenging book to make it entirely through. If you are willing to take some leaps of faith then you should be able to make it.

First and foremost, absolutely zero Rust background is assumed and every new concept will be explained as it arises.

If you have a background that only includes garbage collected, dynamically typed languages, such as Python, JavaScript, Ruby, PHP, then the biggest hurdle will probably be working with the type system. That being said, you might just find a wondrous joy associated with the tight feedback loop of a compiler telling you all the things you have done wrong. Moreover, the type inference will let you forget about it most of the time. Some of the points around memory safety might seem less exciting to you because all of these languages are also memory safe. The approach

to achieving memory safety is different but the end result is the same. Some topics around pointers and references might seem new, but all of these languages leak those concepts and you probably already understand them just in a different form. For example, if you write the following Python:

```
def someFunc(items = []):
    items.append(1)
    return items

a = someFunc()
b = someFunc()

a.append(2)

print(a)
print(b)
```

you will see `[1, 1, 2]` printed twice. As a diligent Python programmer you know not to use lists as default arguments to functions like this and the reason has to do with values versus references. So even if you don't explicitly think that you are working with pointers, you definitely do use them all the time. Rust has many high level syntactic features that make it feel surprisingly similar to a dynamic language.

If you have a background in functional programming coming from Haskell or the ML family then a lot will feel quite at home. But, the use of mutability and explicit imperative style might be less of your thing. Rust has great functional programming facilities and the type system borrows a lot from these languages. However, Rust is more geared towards giving you a lot of the same safety benefits of functional programming while still writing imperative code. Shared, mutable state is the root of all evil. Functional programming attacks that problem by doing away with mutability. Rust attacks it by doing away with sharing.

If you are coming from C++ then you are in for an easier road in some ways and a much harder one in others. I focus here on C++ as it is more superficially similar, but many points also apply to C. Much of the syntax and most of the concepts will be familiar to you. However, there are a few new concepts, like lifetimes, and a few things that look the same but are not, like references and move semantics.

There are APIs you will find in Rust which you might find to be highly performant, but laughably dangerous if ported to C++. You are correct. However, the borrow checker can make such APIs safe. For example, would you give a reference to a stack allocated piece of data to another thread? Would you store a reference to part of a string in a heap allocated struct? Both those are invitations to disaster in C++. They are trivial to do correctly in Rust thanks to the borrow checker and can be great for performance. The API you might find normal in C++ may not be expressible in safe Rust. That is, the borrow checker may not allow you to compile code you consider correct. You may in fact be correct. However, this is usually an API which is easy to misuse and is only correct with a significant amount of cognitive burden on you.

Hence, coming from C++ might require the most shift in how you think about structuring programs. You are more likely to "fight the borrow checker" because some of the ways Rust wants you to do things are just plain against your instincts.

Rust has a bit of notoriety for having a steep learning curve, but it is actually mostly about unlearning things from other languages. Therefore, having less experience can work in your favor.

Getting your environment setup

This book assumes an installed version of Rust and some associated tooling. The first step in getting setup is to visit the official installation website:

> https://www.rust-lang.org/tools/install

You should be able to follow the instructions to get setup via `rustup`. Rust has a fast release cadence for a programming language with a new version every six week. This means that the particular version as of this writing will be stale by the time you are reading it. However, Rust also puts a strong emphasis on backwards compatibility. Thus, as long as you are using a version of Rust at least as new as when this was written, everything should still work for you. Rust 1.37.0 should be new enough for all the code in this book. Moreover, we are using the 2018 edition exclusively. There is an entire guide[6] dedicated to explaining the editions so we will not cover it in depth here.

[6] https://doc.rust-lang.org/edition-guide/index.html

Rustup

The rustup[7] tool is your one stop shop for managing multiple versions of the Rust compiler on your machine. You can have different versions of the compiler installed next to each other and easily switch back and forth between them. You can install nightly releases to try out new features and then easily switch back to stable for other projects. If you have ever dealt with the absolute madness associated with managing different versions of some languages then you will be delighted at how well `rustup` just works.

One note, for some reason all of the details of rustup can be found in the Github readme[8] for the project. It is pretty easy to use but the command line help frequently fails me.

Cargo

`rustc` is the Rust compiler, and you can invoke it directly, however you will find this rarely to be necessary as the majority of your time will be spent interacting with Cargo. Cargo is a dependency manager and a build system. You use a manifest to specify details of your code and its dependencies and you can then instruct Cargo to build your code and it will take care of the rest. You can have Cargo manage building for other platforms and for quickly type checking via `cargo check`. You use it to run tests via `cargo test` and for countless other tasks.

We will cover code structure later on, but the primary unit is known as a crate. You can depend on other crates and the public repository can be found at crates.io[9]. This is related to Cargo in that there is quite a bit of default work builtin to Cargo for working with crates.io, but it is not absolutely required.

Cargo has its own guide[10] which is a great source of information when you find yourself wondering how to do something with Cargo. You can also always run `cargo help` to answer your questions from the command line.

[7] https://rustup.rs
[8] https://github.com/rust-lang/rustup
[9] https://crates.io/
[10] https://doc.rust-lang.org/cargo/guide/

IDEs, RLS, Editors

The editor support story is getting better and is significantly better than it used to be. A project known as the Rust Language Server[11](RLS) is designed to provide the backend for any editor to interact with the compiler and a tool called Racer[12] which provides faster (but less precise) information than the compiler can. This is a project that conforms to the Language Server Protocol[13](LSP) so that every editor which can act as a LSP client can work with RLS. There is a reference implementation of an RLS specific frontend for Visual Studio Code, so if you are unsure where to start that might be one to try out.

If you have a favorite editor already, like Vim or Emacs, then there are plugins you can use to make working with Rust more comfortable. Personally, I use Vim and a shell for running commands directly with Cargo. This is mostly so that I can move between environments with minimal change to my workflow, and I have found that Rust is amenable to this style. There are some languages which are very hard to work with without autocomplete and Rust has not been like that for me.

Check out the official website[14] for an up to date list of tools.

Clippy

The linter is affectionately named Clippy[15]. Cargo supports an awesome feature where you can install subcommands via rustup so that you can selectively add components to Cargo based on your needs. Clippy can be installed this way by running:

```
rustup component add clippy
```

and then run with:

[11]https://github.com/rust-lang/rls
[12]https://github.com/racer-rust/racer
[13]https://langserver.org/
[14]https://www.rust-lang.org/tools
[15]https://github.com/rust-lang/rust-clippy

```
cargo clippy
```

It provides a bunch of helpful information and is good to run against your code regularly. There are many ways to configure it both at a project level as well as at particular points in your code. Linters are still an under used tool that end up being a big source of bike shedding on larger teams. However using at least something to catch the most egregious issues is better than nothing.

Rustfmt

Rust has an official code formatter called rustfmt[16]. This was a project that started life in the community and eventually got official status. However, it is not as seriously official as gofmt for the Go language. You can configure `rustfmt` based on a couple attributes and there is nothing forcing you to use it. But, you should use it. Automated code formatting is one of the great productivity wins of the past twenty years. Countless engineering hours will no longer be wasted debating the finer points of column widths, tabs versus spaces, etc. Let the formatter do its job and get back to building.

Documentation

The standard library has documentation[17] which you will consult frequently. It is thorough and well-written. I know that typing `d` into my browser's address bar and hitting enter will take me to doc.rust-lang.org.

All crates on crates.io[18] will automatically have its documentation built and available on docs.rs[19] which is an amazing tool for the community. Rust has great facilities for including documentation in your code which is why most crates are quite well documented. One excellent feature is the ability to include code samples in your documentation which is actually checked by the compiler. Thus the code examples in the documentation are never out of date

[16] https://github.com/rust-lang/rustfmt
[17] https://doc.rust-lang.org/std/index.html
[18] https://crates.io
[19] https://docs.rs/

The offical rustdoc book[20] is a great resource for learning about documenting your Rust code.

The Nomicon

Rust has a safe and an unsafe side. You may one day find yourself wondering more about what goes on over on the mysterious, dark unsafe side. Well look no further than the Rustonomicon[21] known colloquially as The Nomicon. This book will give you more guidance about how safe and unsafe interact, what you can and cannot do in unsafe Rust, and most importantly how not to break things when you need to use unsafe. It is highly unusual to need to use unsafe. Even when performance is critical, safe Rust most likely can solve your problem. However, there are instances where you really need to reach for this tool and the Nomicon will be your friend at that time.

[20] https://doc.rust-lang.org/rustdoc/index.html
[21] https://doc.rust-lang.org/nomicon/

Summary

The history of computing is filled with powerful abstractions that let the machine manage complexity and thus frees cognitive load from our minds to be spent on more productive tasks. We moved from machine code to assembly to high level languages like C. Each step had a cost associated with giving up some explicit control and a benefit of increased expressive power per unit of code written. Some of these layers were strictly better, i.e. the cost was so negligible compared to the benefits as to be ignored today.

These layers of abstractions continued to be built with different cost/benefit trade-offs. Writing programs to solve complex tasks is a challenging endeavor. Some constraints on our programs have dictated which of those layers of abstractions we can use based on the various trade-offs. At some point, it became generally accepted wisdom that memory safety must be traded off against performance and control. If you want performance then you have to be close to the metal and that is necessarily unsafe. If you want safety then you must be willing to sacrifice some runtime performance to get it. There were counter points in small programs, but no one has challenged this status quo when applied to programming in the large. That is, until Rust.

You can have speed, no garbage collection and therefore a low memory footprint, and you can have safety. Rust affirms our worst fears: programming is hard, humans are fallible. But Rust also assuages those fears by having our computer handle the tasks that are hard for humans. The concepts behind Rust are just the natural evolution in our history of layered abstractions. Whether Rust succeeds or not, history will look back at this as a turning point where it no longer became acceptable to give up safety.

Making Your First Rust App

Getting started

We are going to build an application in Rust to get a feel for the language and ecosystem. The first step for all new Rust projects is generating a new project. Let's create a new project called `numbers`:

```
cargo new numbers
```

Cargo is the package manager for Rust. It is used as a command line tool to manage dependencies, compile your code, and make packages for distribution. Running `cargo new project_name` by default is equivalent to `cargo new project_name --bin` which generates a *binary* project. Alternatively, we could have run `cargo new project_name --lib` to generate a *library* project.

Binary vs. library

A binary project is one which compiles into an executable file. For binary projects, you can execute `cargo run` at the root of your application to compile and run the executable.

A library project is one which compiles into an artifact which is shareable and can be used as a dependency in other projects. Running `cargo run` in a library project will produce an error as cargo cannot figure out what executable you want it to run (because one does not exist). Instead, you would run `cargo build` to build the library.

There are different formats which the Rust compiler can generate based on your configuration settings depending on how you wish to use your library.

The default is to generate an `rlib` which is a format for use in other Rust projects. This allows your library to have a reduced size for further distribution to other Rust

projects while still being able to rely on the standard library and maintain enough information to allow the Rust compiler to type check and link to your code.

Alternative library formats exist for more specialized purposes. For example, the `cdylib` format is useful for when you want to produce a dynamic library which can be linked with C code. This produces a `.so`, `.dylib`, or `.dll` depending on the target architecture you build for.

The generated project

Let's enter the directory for our newly generated Rust project to see what is created:

```
cd numbers
```

The generated structure is:

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

Rust code organization relies primarily on convention which can be overridden via configuration, but for most use cases the conventions are what you want.

main.rs

For a binary project, the entry point is assumed to be located at `src/main.rs`. Furthermore, inside that file, the Rust compiler looks for a function named `main` which will be executed when the binary is run. Cargo has generated a `main.rs` file which contains a simple "Hello, world!" application:

src/main.rs

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

The syntax here says define a function (`fn`) with the name `main` which takes zero arguments and returns the empty tuple `()`.

> Leaving off the return type is equivalent to writing `-> ()` after the argument list of the function. All function calls are expressions which must return a value. The empty tuple `()` is a marker for no value, which is what a function with no return type implicitly returns.

The body of the function calls a macro `println` which prints its argument `"Hello, world!"` to standard out followed by a newline.

> We will cover macros more later, but for now we will mention a few basics. We know it is a macro invocation and not a normal function call because of the trailing `!` in the name. Macros are a powerful form of meta-programming in Rust which you will use frequently but probably rarely find the occasion to have to write. Rust implements `println` as a macro instead of as a regular function because macros can take a variable number of arguments, while a regular function cannot.

The syntax of Rust is superficially similar to C++ which we can see as curly braces are used for denoting blocks and statements are semicolon terminated. However, there is quite a bit more to the Rust grammar that we will cover as we go along.

Caro.toml

The `Cargo.toml` file is the manifest file for the project which uses the TOML[22] format. This is the entry point for describing your project as well as specifying

[22] https://github.com/toml-lang/toml

dependencies and configuration. The initial generated file contains the bare essentials for describing your project:

Cargo.toml

```
[package]
name = "numbers"
version = "0.1.0"
authors = ["Your Name <your.name@example.com>"]
edition = "2018"

[dependencies]
```

The blank section for dependencies is included because nearly every project includes some dependencies. One feature of Rust has been to keep the core language and standard library relatively slim and defer a lot of extra functionality to the community. Therefore relying on third party dependencies is encouraged.

Crates

The primary unit of code organization in Rust is called a crate. Your code exists as a crate which can be distributed to the community via crates.io[23]. Crates in Rust are analogous to gems in Ruby or packages in JavaScript. The registry at crates.io is similar to rubygems.org or npmjs.com as the de facto community repository for distributing and sharing code.

Binary Rust projects are also called crates so they do not solely represent shared library code. Furthermore, a crate can contain both a library and an executable.

It is often difficult to foresee how other's will want to use your software. A common practice in the Rust community is to create dual library/binary crates even when the primary intention of a project is to produce an executable. This can have positive effects on the API design of your code knowing that it should be suitable for external consumption. The binary part of the crate is typically responsible for argument parsing and configuration, and then calls into the functionality exposed by the library

[23] https://crates.io/

part of the crate. Writing all of your code only as an executable and then trying to extract a library after the fact can be a more difficult process. Moreover, the cost of splitting code into a library is minimal.

Making our crate a library

Cargo assumes the entry point for defining a library crate is a file `src/lib.rs`. Let's convert our current binary crate into a binary and library crate. First, we create our library entry point:

src/lib.rs

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

There are two differences to this code from what was in `main.rs`. First, we changed the name of the function from `main` to `say_hello`. This change is more cosmetic than anything (in fact leaving it named `main` works just fine, `main` is only special in some contexts).

The second change is the keyword `pub` before `fn`. This is a privacy identifier which specifies that this function should be publicly accessible to user's of our crate. Without the keyword, we could call this function inside of our `lib.rs` file, but user's of our crate would not be able to call it. Note that our executable sees the library crate the exact same as someone who included our library as a dependency in their `Cargo.toml` file. This ensures a proper separation of concerns between code meant to be executed as a binary and the actual functionality of your project.

We can now change our `main` function to use the functionality exposed by our library:

src/main.rs

```
fn main() {
    numbers::say_hello();
}
```

Running this code should result in the same output as before:

```
$ cargo run
   Compiling numbers v0.1.0 (...)
    Finished dev [unoptimized + debuginfo] target(s) in 0.53s
     Running `target/debug/numbers`
Hello, world!
```

Let's unpack this function call syntax a little bit before moving on. Even though our binary exists in the same codebase as our library, we still must refer to the functions in the crate by the name of the crate, `numbers` in this case.

We wish to call a function named `say_hello` which exists in the `numbers` crate. The double colon operator `::` is used for separating items in the hierarchy of modules. We will cover modules later, but suffice it to say that crates can contain modules, which themselves can contain more modules.

To resolve an item, be it a type or function, you start with the name of the crate, followed by the module path to get to the item, and finally the name of the item. Each part of this path is separated by `::`. For example, to get a handle to the current thread you can call the function `std::thread::current`. The crate here is `std` which is the standard library. Then there is a module called `thread`. Finally inside the `thread` module there is an exported function called `current`.

Items can exist at the top level of a crate (i.e. not nested in any modules), which you refer to simply by the name of the crate, then `::`, then the name of the item. This is what is happening with `numbers::say_hello` because `say_hello` exists at the top level of our `numbers` crate.

Trade-offs

Two of the big selling points of Rust are performance and reliability. Performance meaning both runtime speed and memory consumption. Reliability here means catching bugs at compile time and preventing certain classes of errors entirely through language design. These goals are often seen as classically at odds with one another. For example, C lives in a world where performance is of utmost importance, but reliability is left as an exercise for the implementor.

Rust has no garbage collector and no runtime in the traditional sense. However, most difficulties of working with manual memory management are taken care of for you by the compiler. Therefore, you will often hear "zero cost" being used to describe certain features or abstractions in the language and standard library. This is meant to imply that neither performance nor reliability has to suffer to achieve a particular goal. You write high level code and the compiler turns it into the same thing as the "best" low level implementation.

However, in practice, what Rust really gives you is the tools to make choices about what trade-offs you want to make. Underlying the design and construction of all software is a series of trade-offs made knowingly or implicitly. Rust pushes more of these trade-offs to the surface which can make the initial learning period seem a bit more daunting especially if you have experience in languages where many trade-offs are implicit.

This will be a topic that permeates this book, but for now we will highlight some of these aspects as we make our `numbers` crate do something more interesting.

Print a list of numbers

Let's build an application that creates a list of numbers and then prints each number on a line by itself to standard out. As a first step, let's just say we want to print the numbers one through five. Therefore, our goal is the following:

```
$ cargo run
1
2
3
4
5
```

Let's change our `main` function to call the yet to be defined library function `print`:

src/main.rs

```
fn main() {
    numbers::print();
}
```

Since we want to print one through five, we can create an array with those numbers and then print them out by looping over that array. Let's create the function `print` in `lib.rs` to do that:

src/lib.rs

```
pub fn print() {
    let numbers = [1, 2, 3, 4, 5];
    for n in numbers.iter() {
        println!("{}", n);
    }
}
```

Let's unpack this from the inside out. We have already seen the `println` macro, but here we are using it with a formatted string for the first time. There are two main features of string interpolation in Rust that will take you through most of what you need. The first argument to one of the printing macros (`print`, `println`, `eprint`, `eprintln`) is a double quoted string which can contain placeholders for variables. The syntax for placeholders to be printed "nicely" is `{}`, and for debugging purposes is `{:?}`. The full syntax for these format strings can be found in the official docs[24].

[24] https://doc.rust-lang.org/std/fmt/

The "nice" format is possible when a type implements the `Display` trait. The debugging format is possible when a type implements the `Debug` trait. Not all types implement `Display`, but the standard practice is for all public types to implement `Debug`. So when in doubt, use `{:?}` to see the value of some variable and it should almost always work.

> We will cover traits in detail later, but we will give a crash course here for what is necessary. Traits are part of the type system to mark certain facts about other types. Commonly they are used to define an interface to a particular set of functions that the type in question implements. You can define your own traits as well as implement traits. Whether a type implements a trait must be stated explicitly in code rather than implicitly by satisfying the functional requirements of the trait. This is one of a few differences between Rust traits and Go interfaces.

We will see when creating types later that usually you can get a `Debug` implementation derived for free, but you must implement `Display` yourself if you want it. Most built-in types implement `Display`, including integers, so we use the format string `"{}"` to say expect one variable. Note that the following does not work:

```
println!(n);
```

The first argument to the print macros must be a literal string, it cannot be a variable, even if that variable points to a literal string. Therefore, to print out a variable you need to use the format string `"{}"` as we are doing. If you forget this the Rust compiler will suggest that as what you probably want to do.

Iteration

So assuming that `n` holds an integer from our collection, we are printing it out using the `println` macro. How does `n` get bound to the values from our collection? We loop over our collection using a `for` loop and bind `n` to each value. The syntax of a `for` loop is:

```
for variable in iterator {
  ...
}
```

Note that we are calling the method `iter` on our array. Rust abstracts the idea of iteration into yet another trait, this one called `Iterator`. We have to call `iter` here to turn an array into an `Iterator` because arrays do not automatically coerce into into an `Iterator`. We shall see shortly that this is not always necessary with other collections.

This is also the first time we are calling a method on an object. Rust types can implement functions that operate on themselves and can therefore be called using this dot syntax. This is syntactic sugar for a direct function call with the receiver object as the first argument. We will cover how these functions are defined when we construct our own types and implement methods on them.

Defining Array Types

We can move out further now to the definition of our array. Rust borrows many ideas of the ML family of languages so some concepts might be familiar if you have experience in that area. By default variables are immutable. Therefore we declare an immutable variable called `numbers` which is bound to an array with the numbers we are interested in. Rust infers the type of `numbers` based on the value we used to initialize the variable. If you want to see the type that is inferred by Rust, a trick is to write:

```
let () = numbers;
```

after the line that declares the variable `numbers`. When you try to compile this code, there will be a type mismatch in the assignment which will print out what the compiler expects:

```
$ cargo run
   Compiling numbers v0.1.0 (...)
error[E0308]: mismatched types
 --> src/lib.rs:3:9
  |
3 |     let () = numbers;
  |         ^^ expected array of 5 elements, found ()
  |
  = note: expected type `[{integer}; 5]`
             found type `()`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
error: Could not compile `numbers`.

To learn more, run the command again with --verbose.
```

We see that the compiler inferred a type of `[{integer}; 5]` for `numbers`. Arrays in Rust are a homogeneous container (all elements have the same type) with a fixed size. This allows it to be stack allocated. The ability to ensure data is stack allocated rather than heap allocated is one of the areas in which Rust allows you to decide what trade-offs you want to make. On the other hand, because an array has a fixed size that must be known at compile time it is not useful for data which might need to grow or shrink or contain an unknown numbers of items. For this we have the `Vec` type which we will return to shortly.

You can also see that the compiler infers the type of elements of the array to be `{integer}` which is a placeholder as without any further constraints the specific type of integer is unknown. Rust has twelve integer types which depend on size and whether it is signed or unsigned. The default is `i32` which means a signed integer that takes 32 bits of space. The equivalent unsigned type is `u32`. Let's say we wish our numbers to be `u8`, that is 8-bit unsigned integers. One way to do this is to specify directly on the numerical constant what type we want:

```
let numbers = [1u8, 2, 3, 4, 5];
```

If we do this then the compiler will infer the type `[u8; 5]` for our array. The other way is to explicitly write out the type of the variable `numbers`:

```
let numbers: [u8; 5] = [1, 2, 3, 4, 5];
```

Type annotations are written with a colon after the variable name followed by the type. We see that the size of the array (5) is part of the type. Therefore, even with the same type of elements, say `u8`, an array with four elements is a different type than an array with five elements.

Using std::vec::Vec

Rust provides a few mechanisms for alleviating some of the limitations of arrays. The first we will talk about is the vector type in the standard library, `std::vec::Vec`[25]. A vector is similar to an array in that it stores a single type of element in a contiguous memory block. However, the memory used by a vector is heap allocated and can therefore grow and shrink at runtime. Let's convert our library print function to use a vector:

src/lib.rs

```
1 pub fn print() {
2     let numbers = vec![1, 2, 3, 4, 5];
3     for n in numbers {
4         println!("{}", n);
5     }
6 }
```

We are calling another macro `vec` which this time constructs a vector with the given values. This looks very similar to the array version, but is actually quite different. Vectors own their data elements, they have a length which says how many elements are in the container, and they also have a capacity which could be larger than the

[25] https://doc.rust-lang.org/std/vec/struct.Vec.html

length. Changing the capacity can involve quite a bit of work to allocate a new region of memory and move all of the data into that region. Therefore, as you add elements to a vector, the capacity grows by a multiplicative factor to reduce how frequently this process needs to take place. The biggest advantage is that you do not need to know upfront how large the vector needs to be; the length is not part of the type.

The type of a vector is `Vec<T>` where `T` is a generic type that represents the types of the elements. Therefore, `Vec<i32>` and `Vec<u8>` are different types, but a `Vec<u8>` with four elements is the same type as one with five elements.

Note also that we are no longer explicitly calling `iter` on the `numbers` variable in our for loop preamble. The reason for this is that `Vec` implements a trait that tells the compiler how to convert it into an iterator in places where that is necessary like in a for loop. Calling `iter` explicitly would not be an error and would lead to the same running code, but this implicit conversion to an iterator is common in Rust code.

Function Arguments

Let's abstract our print function into two functions. The entry point will still be print (so we don't need to change `main`) which will construct a collection, but it will then use a helper function to actually print the contents of this collection. For now we will go back to using an array for the collection:

src/lib.rs

```
pub fn print() {
    let numbers = [1, 2, 3, 4, 5];
    output_sequence(numbers);
}

fn output_sequence(numbers: [u8; 5]) {
    for n in numbers.iter() {
        println!("{}", n);
    }
}
```

This is our first function that has input or output. Type inference does not operate on function signatures so you must fully specify the types of all inputs and the output.

However, we still are not returning anything so by convention we elide the `-> ()` return type which is the one exception to the rule of fully specifying the types in function signatures.

The input type of our function `output_sequence` is our five element array of `u8` values.

Rust has a few different modes of passing arguments to functions. The biggest distinction being that **Rust differentiates between**:

- a function temporarily having access to a variable (borrowing) and
- having *ownership* of a variable.

Another dimension is whether the function can mutate the input.

The default behavior is for a function to take **input by value and hence ownership** of the variable is moved into the function.

The exception to this rule being if the type implements a special trait called `Copy`, in which case the input is copied into the function and therefore the caller still maintains ownership of the variable. If the element type of an array implements the `Copy` trait, then the array type also implements the `Copy` trait.

Suppose we want to use a vector inside `print` instead, so we change the code to:

src/lib.rs

```
pub fn print() {
    let numbers = vec![1, 2, 3, 4, 5];
    output_sequence(numbers);
}

fn output_sequence(numbers: [u8; 5]) {
    for n in numbers.iter() {
        println!("{}", n);
    }
}
```

But this won't work because `[u8; 5]` and `Vec<u8>` are two different types. One possible fix is to change the input type to `Vec<u8>`:

src/lib.rs

```rust
pub fn print() {
    let numbers = vec![1, 2, 3, 4, 5];
    output_sequence(numbers);
}

fn output_sequence(numbers: Vec<u8>) {
    for n in numbers {
        println!("{}", n);
    }
}
```

This works for this case. It also let's us see what happens when passing a non-Copy type to a function. While arrays implement the `Copy` trait if their elements do, `Vec` does not. Hence, try adding another call to `output_sequence(numbers)` after the first one:

src/lib.rs

```rust
pub fn print() {
    let numbers = vec![1, 2, 3, 4, 5];
    output_sequence(numbers);
    output_sequence(numbers);
}

fn output_sequence(numbers: Vec<u8>) {
    for n in numbers {
        println!("{}", n);
    }
}
```

This gives us an error:

```
$ cargo run
   Compiling numbers v0.1.0 (...)
error[E0382]: use of moved value: `numbers`
 --> src/lib.rs:4:21
  |
3 |     output_sequence(numbers);
  |                     ------- value moved here
4 |     output_sequence(numbers);
  |                     ^^^^^^^ value used here after move
  |
  = note: move occurs because `numbers` has type `std::vec::Vec<u8>`, w\
hich does not implement the `Copy` trait

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
error: Could not compile `numbers`.

To learn more, run the command again with --verbose.
```

We can see Rust generally has very helpful error messages. The error is that a value was used after it has been moved. The `print` function no longer owns `numbers`. The "note" in the error explains why the move happens due to vector not implementing the `Copy` trait.

Note that in the changes we have made, the body of `output_sequence` has remained the same (modulo whether we call `iter` explicitly or not), only the type signature has been changing. This is a hint that maybe there is a way to write a type signature that works for both arrays and vectors. There are again several ways to accomplish this goal.

A type signature that works for both arrays and vectors

As we have said before, Rust has a lot of power and gives you very fine-grained control over what you want to use or don't want to use. This can be frustrating when starting out because any time you ask "what is the right way to do this," you

will invariably be met with the dreaded "it depends." Rather than detail every possible permutation that achieves roughly the same outcome, we are going to focus on the most common idioms. There are certain performance reasons as well as API design decisions that lead to different choices, but those are more exceptional cases than the norm. We will provide pointers to the choices we are making when it matters, but note that due to the scope of the language there is almost always more than one way to do it.

A key type that comes in handy to alleviate some of the limitations of arrays is the `std::slice`[26]. Slices are a dynamically sized view into a sequence. Therefore, **you can have a slice which references an array or a vector and treat them the same**. This is a very common abstraction tool used in Rust. This will be more clear by seeing this in action.

Let's change the signature of `output_sequence` to take a reference to a slice, and change `print` to show that it works with both arrays and vectors:

src/lib.rs

```
pub fn print() {
    let vector_numbers = vec![1, 2, 3, 4, 5];
    output_sequence(&vector_numbers);
    let array_numbers = [1, 2, 3, 4, 5];
    output_sequence(&array_numbers);
}

fn output_sequence(numbers: &[u8]) {
    for n in numbers {
        println!("{}", n);
    }
}
```

A slice of `u8` values has type `[u8]`. This represents a type with an unknown size at compile time. The Rust compilation model does not allow functions to directly take arguments of an unknown size. In order to access this slice of unknown size with something of a known size we use indirection and pass a reference to the slice

[26] https://doc.rust-lang.org/std/slice/index.html

rather than the slice itself. A reference to a slice of `u8` values has type `&[u8]` which has a known size at compile time. This size is known because it is equal to the size of a pointer plus the length of the slice. Note that slices convert automatically into iterators just like vectors so we again do not call `iter` explicitly in the body of our function. This takes care of the signature of `output_sequence` however the way we call this function from `print` has changed as well.

Notice that we have added an `&` before the variable names that are passed to `output_sequence`. You can think of this as creating a slice that represents read-only access to the entire sequence for both the vector and array. However, this small change in how we call the function allows us to handle vectors and arrays equally well. Idiomatic Rust takes slices as arguments in most cases where one needs only to read the collection. This is particularly true for strings which we will cover later.

The major difference here is that we are no longer transferring ownership into the function `output_sequence` instead we are lending read-only access to that function. The data is only borrowed for the duration of the function call. The idea of ownership and borrowing is a core part of the Rust language and is something we will be constantly running into.

Constructing A Vector of Numbers

Let's make one more change to make this program more flexible. Instead of printing out one through five, let's take a number as input and print from one up to that value. We could just iterate through integers and print them out as we go along rather than using the `output_sequence` helper function. However, we are going to construct a vector to show a few more language features.

Let's create yet another helper function, `generate_sequence` which takes a limit as input and outputs a vector. Our `print` function can then just combine these two parts:

src/lib.rs

```rust
pub fn print(limit: u8) {
    let numbers = generate_sequence(limit);
    output_sequence(&numbers);
}

fn generate_sequence(limit: u8) -> Vec<u8> {
    let mut numbers = Vec::new();
    for n in 1..=limit {
        numbers.push(n);
    }
    numbers
}

fn output_sequence(numbers: &[u8]) {
    for n in numbers {
        println!("{}", n);
    }
}
```

In `print` we bind a variable to the result of calling `generate_sequence` with the `limit` passed to us as the argument, then we call `output_sequence` as before passing a reference to a slice backed by the variable we just created.

The new function here takes an input argument, `limit`, and returns a `Vec<u8>`. This is our first function returning something.

Again as there are a lot of different ways to do things in Rust, we are going to just show one particular way to construct the vector we desire in order to hit some relevant parts of the language.

First we create a new vector with `Vec::new()`.

Unlike in some other languages, `new` is not special but rather has become by convention the name of the function that returns a new instance of a type. You can write a function called `new` which does something else and it would compile just fine, but it would go against the standard way of doing things.

By default a vector created with `new`, is the same as one created with `vec![]`, and does not allocate. Therefore, unless you actually put something into a vector it does not use any memory.

In the code above, we see a new keyword being used, `mut`. Mutability is a property of the variable or reference not of the object itself. Therefore we declare `numbers` to be a mutable variable that holds an empty vector. This allows us to later call `numbers.push(n)` because `push` is a method that requires the receiver to be mutable. Removing the `mut` from the `let` binding will result in a compiler error when we try to `push`.

In order to generate the numbers starting at `1` up to our limit, we use a `for` loop, but this time the iterator is a `Range`[27] object, in particular an `InclusiveRange`. Ranges can be constructed with using the syntax `start..end` or `start..=end`. Both `start` and `end` are optional, and if you have neither, i.e. `..`, then you also cannot have the `=` sign. By default the range is inclusive on the left (i.e. includes `start`), and exclusive on the right (i.e. does not include `end`). The `=` after the two dots makes it so the range includes the end point. We want the numbers starting at `1` up to `limit`, including the `limit`, so we use `1..=limit`. Ranges are frequently used when creating slices, for example:

```
let numbers = [1, 2, 3, 4, 5];
let subset = &numbers[1..3];
```

Here `subset` is a slice of length `3-1=2` which starts at index `1`, hence it is the slice `[2, 3]`.

Iterating over this range, we `push` each value onto the end of our vector which causes heap allocations every time there is not enough capacity to extend the length. Finally, we want to return this vector from our function. The final expression in a function is

[27] https://doc.rust-lang.org/std/ops/struct.RangeInclusive.html

implicitly returned so there is no need for an explicit `return` statement. However note the lack of semicolon at the end of the last line of this function. The expression that evaluates to the vector `numbers` is written without a semicolon and means to return that value. If we had written a semicolon, that would be a statement whose value is `()` which is not what you want to return. This is a common error so the compiler is smart enough to tell you what to fix, but it is nonetheless an error. You can use a `return` statement to return early from a function, but using the last expression of the block as the implicit return is idiomatic Rust.

A Shorter Version with collect

The purpose of writing `generate_sequence` like this was to demonstrate object construction, mutability, and ranges. Before leaving, let's look at a very powerful construct that is used throughout real Rust which would be a better approach to generating this vector. We could replace `generate_sequence` with:

src/lib.rs

```
6  fn generate_sequence(limit: u8) -> Vec<u8> {
7      (1..=limit).collect()
8  }
```

Rust has powerful generic programming facilities which allows for the function `collect` to exist. This function can be used to turn any iterator into basically any collection.

Commonly, one takes a collection like a vector, turns it into an iterator by calling `iter`, performs transformations on the generic iterator, and then calls `collect` at the end to get back whatever specific collection one wants to work with. It can also be used to directly turn one collection into another collection, which is what we are doing here by turning a range into a vector.

Collect is a generic function over the return type, so the caller gets to determine what they want. Here because we return the result of calling `collect` from our function, type inference sees that the return type needs to be a `Vec<u8>` and therefore ensures that `collect` generates that collection. While the type inference in Rust is good, it some times cannot figure out what you want when using `collect`. Therefore, you

might find the need to use the syntax `collect::<SomeType>()` to help the compiler know what you want.

This syntax, `::<>`, you may see referred to as the "turbofish".

We have changed our exported `print` function to require an input variable, so we need to update our call in `main` to pass something in. Let's pass in `5` to get the same output as before:

src/main.rs

```
1  fn main() {
2      numbers::print(5);
3  }
```

Testing our code

Testing is a large topic and is something we will cover in more detail as we move to larger applications, however let's write our first test to see how easy it can be in Rust. Add the following to the end of `src/lib.rs`:

src/lib.rs

```
20 #[test]
21 fn generate_sequence_should_work() {
22     let result = generate_sequence(3);
23     assert_eq!(result, &[1, 2, 3]);
24 }
```

Our test is just a normal function with a special attribute, `#[test]`, before it. We will see attributes come up frequently as they are used for a variety of purposes in Rust. They come in two forms `#[...]` and `#![...]` which annotate the item they precede. The name of the test comes from the name of the function. Inside of test functions

there are a series of macros you can use for asserting facts about your system. We use `assert_eq` to ensure that the output of our `generate_sequence` function is what we expect it to be.

We can use `cargo` to run all of our tests:

```
$ cargo test
    Finished dev [unoptimized + debuginfo] target(s) in 0.43s
     Running target/debug/deps/numbers-f74640eac1a29f6d

running 1 test
test generate_sequence_should_work ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered \
out
```

Wrapping up

The Rust language is designed to allow fine grained control over performance and reliability while writing code at a high level of abstraction. The cost is thus learning the abstractions and dealing with the cognitive load of making choices as you design a program. These considerations are important for production quality code as you profile and optimize where you find bottlenecks. However, following the standards of the community will get you the majority of the way to high quality applications.

We will not belabor all of the nuance in the finer points of making every decision along the way. Our goal is to provide a solid foundation upon which you can explore alternatives as necessary by using the standard library documentation.

Making A Web App With Actix

Web Ecosystem

One area where Rust stands out is in the building of web servers.

Rust has its origins at Mozilla primarily as a tool for building **a new browser engine**. The existing engine being written in C++ combined with the syntactical similarities encourages the idea the Rust was meant to be a replacement for C++. There is obviously some truth to this, but in many ways this characterization sells Rust's potential short. While it is capable of being a systems programming language, there are a plethora of language features that make it suitable for innumerable programming tasks, including building web servers.

There are a few different layers to the web programming stack. Primarily we are concerned here with the application layer which is comparable to where Django, Rails, and Express live in Python, Ruby, and NodeJS, respectively.

The ecosystem around web application development in Rust is still quite nascent despite the Rust language hitting 1.0 in 2015. Much of the underlying infrastructure for building concurrent programs took until 2019 to reach a maturity sufficient to be included in the stable version of the standard library. However, most of the ecosystem has coalesced around similar ideas which take advantage of Rust's particular features.

Before jumping in to building a simple web server, let's briefly discuss a few of the libraries that make up the web landscape.

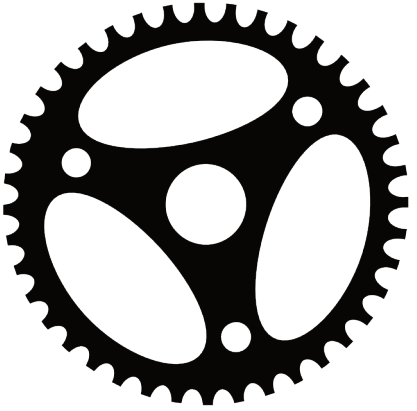
Hyper

Hyper[28] is a low level HTTP library built on even lower level libraries for building network services. Currently most web frameworks use Hyper internally for handling the actual HTTP requests.

[28] https://hyper.rs/

It can be used to build both HTTP clients and servers. However, there is a bit more boilerplate than you might want to write yourself when you want to focus on building an application. Therefore, we will use a library at a higher level of abstraction which still allows us to take advantage of what Hyper offers.

Actix



Actix

The Actix[29] project is actually a group of projects which define an actor system as well as a framework for building web applications. The web framework is aptly named `actix-web`. It has been built on top of futures and async primitives from the beginning. It also runs on the stable version of the compiler.

It recently hit the 1.0 milestone which should bring some much needed stability to the ecosystem. Additionally, it has been at the top of the Tech Empower web framework benchmarks[30]. Even if those are artificial benchmarks, it still points to the performance potential possible.

Actix is the library that we are going to use in this chapter, but before we dive in, let's look at a few others from a high level.

[29] https://actix.rs/
[30] https://www.techempower.com/benchmarks/#section=data-r16&hw=ph&test=plaintext

Rocket

Rocket

Rocket[31] is a web framework which is easy to use while maintaining flexibility, speed, and safety. One of the downsides to this is that currently Rocket only works with the nightly compiler. This is mostly because certain compiler features that make the application developer's life better are not available on stable yet.

Moreover, Rocket intends to move to an async backend once those ideas stabilize in the standard library, but as of version 0.4 the backend is still a synchronous design.

Finally, Rocket has not yet reached 1.0 as there is still a lot of work to do to which might necessitate breaking changes. It is a high quality library which will not likely have much breakage going forward, but there is still some risk which depends on your tolerance.

Others

There are several other web frameworks in the Rust ecosystem and but it's not possible to cover everything here. Some promising libraries that were built up have fallen into an unmaintained state, whereas others are still building up and are explicitly not ready for prime time.

[31] https://rocket.rs/

Iron

Iron[32] is arguably the original Rust web framework. It was actively maintained for quite a while, then was partially abandoned for a while, and then recently resurrected and updated. Iron takes a different approach than the other web frameworks and is more similar to frameworks in other languages.

It is built to have a small, focused core and then provide the rest of its functionality via plugins and middleware. This style is possibly more familiar to web developers coming from other languages, but in the Rust ecosystem, at least currently, this is not the norm.

The core Rust language has an Async Working Group that has an experimental web framework called Tide[33]. They are working out how the async primitives in the standard library can be used to build a web framework that is easy to use and safe. It is explicitly not ready for production, but many of the ideas are similar to what is being done in Actix and Rocket.

Starting out

We are going to use `actix-web` because it works with the stable compiler. A lot of the underlying ideas about how data flows in the system is similar to how Rocket works, so switching between the two should not be a fundamental shift in how the application is built.

So let's get started with a new project:

[32] http://ironframework.io/
[33] https://github.com/rustasync/tide

```
cargo new messages-actix
```

We let `cargo` bootstrap our project as we have done before. The first thing we are going to do is edit our `Cargo.toml` file to add all of the dependencies that we will use:

Cargo.toml

```
[package]
name = "messages-actix"
version = "0.1.0"
authors = ["Your Name <your.name@example.com>"]
edition = "2018"

[dependencies]
actix-web = "1.0"
env_logger = "0.6"
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

We have already talked about our first dependency, `actix-web`. The second dependency listed, `env_logger`, will be used to allow us turn on/off the logging features of actix.

We import `serde` which is the de facto standard way of implementing serialization and deserialization between native Rust types and a variety of formats. The main serde crate takes care of much of the generic machinery, and then outsources the specifics of different formats to other crates. Therefore, to enable marshalling to/from JSON, we bring in the `serde_json` crate.

The name `serde` is a portmanteau of *ser*ialize and *de*serialize.

Dependencies can take a variety of forms depending on the level of configuration desired. The typical format is:

```
crate_name = "X.Y.Z"
```

where `crate_name` is the official name for the crate as published on crates.io and the string value "X.Y.Z" is a semver[34] requirement which is interpreted the same as "^X.Y.Z". There exists a lot of power in specifying exactly how you want your dependencies to update, but the most common form is `crate_name = "X"`, where `X` is the major version that your code is compatible with.

Getting the structure in place

As discussed previously, we will split up our crate into both a library and a binary by putting most of our code in `lib.rs` or other further modules, and then calling into that library from `main.rs`. Therefore, let's get our `main.rs` file setup once and then all of our changes will be in the library:

src/main.rs

```
use messages_actix::MessageApp;

fn main() -> std::io::Result<()> {
    std::env::set_var("RUST_LOG", "actix_web=info");
    env_logger::init();
    let app = MessageApp::new(8080);
    app.run()
}
```

We will create and export a type called `MessageApp` which will be the entry point into our library, so our first step is to import this type.

[34] https://github.com/steveklabnik/semver#requirements

It may be initially confusing that the name of our library as specified in `Cargo.toml` is `messages-actix` (with a hyphen), and yet in main we are importing the type `MessageApp` from the crate `messages_actix` (with an underscore). Crate names are allowed to contain hyphens and underscores, but identifiers in Rust are not allowed to contain hyphens. Therefore, if you use a crate name with an underscore the compiler knows to look for a crate with a hyphen if one with an underscore cannot be found.

The community seems to not have completely coalesced on choosing one style for crate names as can be noted by comparing `actix-web` with `serde_json`. If you use an underscore in your crate name instead of a hyphen then this translation step is unnecessary.

We define the `main` function which will be entry point for our binary. As of Rust 1.26 it is possible to for the `main` function to return a `Result` type. `Result` is one of the primary error handling primitives that Rust provides. Let's take a moment to talk about Rust's facilities for abstraction.

Aggregate data types

The primary mechanisms for creating aggregate data types in Rust are **enums** and **structs**. In theoretical terms, these are aggregates of other types where structs represent product types and enums represent sum types. Enums in Rust can contain data and are therefore related to algebraic data types in functional languages.

We will return to structs once we get to our library code, so let's focus on enums for the moment by looking at `Result` as an example. `Result` is an enum that is defined in the standard library as:

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

There are a lot of new things happening here so we will go through it step by step. Enums represent a type where we enumerate the different possible values that a particular value represents. These are sometimes called tagged unions. The idea is

that a value of type `Result` can be in exactly one of two states: the `Ok` variant or the `Err` variant. Enum variants can contain data like we see here, or they can be just a name, for example:

```
enum Color {
  Red,
  Green,
  Blue,
}
```

Result

`Result` is also an example of a generic type because it is parameterized by two type variables `T` and `E` which we can see in angle brackets after the name of the type. These generic types are used in the two variants as a way of encapsulating data along with the identity of the variant. Generic types can have restrictions placed on them, but here there are none so `T` and `E` can be any type. The names of variants can be referred to just like other items that live within some outer structure using the `::` operator. Therefore, `Result::Ok(true)` would construct the `Ok` variant of the type `Result<bool, E>` where `E` would need to be further specified by the context. For our `Color` enum, we can refer to variants like `Color::Red` or `Color::Green`.

`Result` is a special type that is available everywhere in your code because normal Rust code works as if a few things are automatically imported at the top of your file. We will see `Result` used throughout all of the code we write in Rust because it allows us to handle success and failure cases in a structured and cohesive way. This is one of the tools that replace situations where one might use exceptions in another language.

Going back to our main function, we are returning the type `std::io::Result<()>`. Within the `std::io` module there is a type definition:

```
type Result<T> = Result<T, std::io::Error>;
```

The type `std::io::Error` is a struct defined in that module for representing errors from I/O operations. The syntax for the type definition above just says to create an

alias `std::io::Result<T>` which uses the standard `Result` type with the second type parameter (which represents the type inside the error variant) fixed to the one defined in the `std::io` module. Therefore, `std::io::Result<()>` is the same as `Result<(), std::io::Error>`.

A type aliases like this does not create a new type but rather just allows for more convenient syntax. It is very common to see type aliases used to fix some of the generic parameters in a type. In particular, you are likely to see this exact kind of type alias used in many libraries because writing `Result<T>` is much more convenient than `Result<T, some::crate::Error>`.

Ok

The `Ok` variant has type `()` which is known as the empty tuple which we have seen before. This is commonly used as a marker where you don't actually care about the value but you need some placeholder type. This `Result` type basically means that you can tell the difference between a success and a failure, and if there is a failure you will get an error that can tell you what happened, but in the success case there isn't anything interesting to add besides the fact that there was a success. This pattern is similar to what we see in C with functions that return `0` on success and non-zero on failure where each non-zero value maps to some specific type of failure.

We use this type as the result of main because our web server will be doing I/O operations and as we will see our server returns this type to signify success or failure. Rust will return an error code and print the debug representation of the error if the returned result of main is the error variant, otherwise it will just exit normally. Hence as any data in the `Ok` variant would go unused anyway, it is natural to use the empty tuple as a marker of success.

Basic logging setup

The first line of our main function sets an environment variable, `RUST_LOG`, to the value `actix_web=info`. We do this for convenience in this program so that we don't have to set the environment variable in our terminal, but normally you would set this in your environment.

Most crates in Rust use the `log`[35] crate to allow for a clean separation between log statements and where they are printed. The `log` crate provides a set of macros that libraries can use to make logging statements at various levels of verbosity. For example:

```
pub fn do_work(num: u32) {
  trace!("About to do work for num: {}", num);
  ...
}
```

Those logging statements do not actually do anything unless a program is configured with an implementation. We choose to use the implementation provided by the `env_logger` crate which we turn on with the call to `env_logger::init()`. There exist a variety of implementations that do things like add colors to the output or send statements to files. We are going to use this simple implementation that just prints statements to standard out based on the settings given in the `RUST_LOG` environment variable.

Starting the app

Finally, the last two lines of our main function are where the real work happens.

src/main.rs

```
6     let app = MessageApp::new(8080);
7     app.run()
```

We create a new instance of our `MessageApp` type by calling `new` with a port number and bind the result to the variable `app`. Then we call the `run` method on our app instance. The lack of a semicolon at the end of this line means that our main function returns the result of the `run` method. Based on the discussion we just had, we therefore know that `run` must return a `Result`.

This main function will stay the same from here on, and we will pack all of the functionality into our `MessageApp` type which need only expose the `new` and `run` methods that we use.

[35] https://docs.rs/log

Handling our first request

We are going to get all of the infrastructure in place to build and run a web server which can handle HTTP requests. Our first incarnation will just support a get request to one route and will respond with a JSON message.

Let's get started with our library implementation by creating `lib.rs` and starting out with some imports:

src/lib.rs

```
#[macro_use]
extern crate actix_web;

use actix_web::{middleware, web, App, HttpRequest, HttpServer, Result};
use serde::Serialize;
```

We are going to use some of the macros exported by `actix_web` which we could import individually in the 2018 edition of Rust, but for now we will import them via the older `#[macro_use]` attribute on the `extern crate` statement.

We import some items from `actix_web` which will make our code a little less verbose. Note in particular that we import `Result` which is the type alias of `Result` that `actix_web` defines with the error type fixed to its error type. We are going to construct a Rust type that represents the data we want to respond to requests with, so we import `Serialize` from `serde` which will allow us to convert that Rust type to JSON data.

Creating our app

We saw in `main` that we imported a struct from our library so let's define that:

src/lib.rs

```
25 pub struct MessageApp {
26     port: u16,
27 }
```

This is the first time we are creating a struct which is the other primary aggregate data type in Rust besides enums.

Structs have *member data* which can be of any type. Here we have one member named `port` of type `u16`. The `pub` specifier before the `struct` keyword means that this type will be publicly exported by our library.

Each member field has its own privacy which is not exported by default. Therefore, even though you can reference instances of type `MessageApp` outside of our library, you cannot directly access the `port` field. We can access the `port` field within the file that defines the type, but otherwise it is hidden.

Similar to enums, structs can also be generic over the types of data they contain. For example, `Vec<T>` which we have seen before is actually a struct called `Vec` which has one generic type parameter.

Adding behavior to our data

Now that we have our struct defined, we can add functionality to it. Rust has a strong separation of data and functionality. We defined the data representation of our struct, but all methods associated with the type are defined elsewhere in what is known as an `impl` block.

These blocks are used for adding functionality to types as well as for implementing traits. All types that you create (structs, enums, etc.) can have functionality added via an `impl` block. Let's work through the `impl` block for `MessageApp`:

src/lib.rs

```
impl MessageApp {
    pub fn new(port: u16) -> Self {
        MessageApp { port }
    }

    pub fn run(&self) -> std::io::Result<()> {
        println!("Starting http server: 127.0.0.1:{}", self.port);
        HttpServer::new(move || {
            App::new()
                .wrap(middleware::Logger::default())
                .service(index)
        })
        .bind(("127.0.0.1", self.port))?
        .workers(8)
        .run()
    }
}
```

A type can have multiple `impl` blocks associated with it, however typically there is only one main one with others usually only for trait implementations. We will return to traits and their implementation at a future point.

Self is special

The first method that we define is called `new` which takes a `port` parameter and returns the special type `Self`. Inside an `impl` block `Self` has special meaning, it refers to the type on which we are defining the implementation. So we could have written the signature of new as:

```
pub fn new(port: u16) -> MessageApp
```

However idiomatic Rust code uses `Self` for such a return type. The name of `new` is not special, but has become convention as the name of the constructor function for types. Therefore, while you could name your constructors anything you want, `new` is what others will expect.

Create a MessageApp

The body of our constructor is pretty simple, we just create a `MessageApp` struct with the data that was passed in. If there is a local variable with the same name and type as a struct field, you can use a shorthand syntax by just writing the variable name in the struct literal. Suppose for example that we wanted to add one to whatever port was passed in to us, then we would have to use the longer syntax:

```
MessageApp {
  port: port + 1,
}
```

To construct a struct we must list each field name followed by a colon followed by the value, unless we can skip the colon and value in the situation described above. Rust allows trailing commas in pretty much every position where a comma could exist in the future and it is standard practice to include them to reduce future diffs when code changes. Again we are returning this newly constructed struct because we do not end this line with a semicolon.

Instance methods

The next method we define is called `run` which takes the special parameter `&self` and returns the `std::io::Result` that we introduced in the previous section.

src/lib.rs

```
34      pub fn run(&self) -> std::io::Result<()> {
35          println!("Starting http server: 127.0.0.1:{}", self.port);
36          HttpServer::new(move || {
37              App::new()
38                  .wrap(middleware::Logger::default())
39                  .service(index)
40          })
41          .bind(("127.0.0.1", self.port))?
42          .workers(8)
43          .run()
```

Inside an `impl` block there are a few different special values which can be the first parameter to functions to signify that those functions are actually instance methods. This is similar to Python where class instance methods explicitly take `self` as their first parameter, and not taking `self` implies that the method is actually on the type rather than a particular instance.

All of the selfs

Due to the semantics around borrowing and mutability, there are four special first parameter values: `&self`, `self`, `&mut self`, and `mut self`. All of the forms turn a function in a method on an instance of the type. This means that rather than being a function on the type which is called like `MessageApp::new`, we need to have constructed an instance of the type and then use dot syntax to call the method and set the first parameter.

In our example, we first call `new` to construct an instance of `MessageApp` called `app`, and then we call `run` as `app.run()`. It is also possible to make this call explicitly as `MessageApp::run(&app)` but that is uncommon.

The first form, `&self`, is the most common form. This means that our method takes an immutable reference to the instance invoking the method. We can read the data inside our type, but we cannot alter it. The calling code also maintains ownership so we are just borrowing the instance.

The second form, `self`, means that the method consumes `self` and therefore the instance that the method is being called on has its ownership moved into the method. This form comes usually when we are transforming a type into something else, for example with interfaces that use the builder pattern.

The third form, `&mut self`, is the mutable version of the first form. This is the second most common thing you will encounter. Our method can read and write the data inside our type, but it does not own the value so this access is only temporary.

Finally the last form, `mut self`, means that this method consumes `self` and `self` is mutable within the method. All parameters to functions can be declared mutable if you wish them to be a mutable binding inside the function, and `self` is no different. This has its uses, but is not that common.

Running our server

The first line of our `run` function just prints out that we are starting a server at a particular address and port. The next expression creates a server, binds it to an address, sets the number of workers, and finally starts the server.

`HttpServer` is the type which `actix-web` exposes to represent something that serves requests. The constructor takes an application factory which is any function that when called returns an application.

We use a closure to define just such a function inline. Closures in Rust can be a little tricky because of the ownership and borrowing semantics. The basic syntax is to declare an argument list between pipes, `||`, then possibly list the return value, followed by the function body between curly braces.

Type inference works on closures so we can usually omit types of the arguments and return values.

Understanding closures

The tricky bit is how the closure captures variables from the surrounding environment, so let's take a closer look at this.

If the keyword `move` comes before the argument list then any variables from the environment that the closure uses are actually moved into the closure. This means the closure takes ownership of those variables rather than creating references.

This implies that the lifetime of the closure can be longer can its surrounding environment because those variables are moved into the closure. Without the `move` keyword, variables closed over are actually just references to the surrounding environment.

Here we do not actually use any variables from the surrounding environment so this example would work without the `move` keyword. However, we will be doing this shortly, so it makes sense to get this out of the way now. Moreover, our intent is for this closure to be entirely owned by the `HttpServer` and therefore the `move` signifies

intent that *the function should not have references to the environment in which it was created.*

Closures are a very nice part of Rust, and they can be hard to fully grasp at first, but this complexity buys performance benefits with control in your hands as to exactly what you want to happen.

Take a look at the relevant part of the code again:

src/lib.rs

```
36          HttpServer::new(move || {
37              App::new()
38                  .wrap(middleware::Logger::default())
39                  .service(index)
40          })
```

Inside the closure, we are construct an `App` which is the abstraction `actix-web` defines for representing a collection of routes and their handlers. We use the `new` method to create an `App`, and then a couple methods defined on that instance to setup our application.

The `wrap` function wraps the app with a middleware specified as its only argument. We set the `Logger` middleware which is provided by actix so that we can see some information about requests as they come in.

Furthermore, we call `service(index)` to specify that we want to add a service to our app which uses the handler `index` which we will define below.

Syntax for working with Results

The last new bit of Rust that we find in this method is the `?` operator after the call to `bind`. The `Result` type is quite special in Rust to the point of having special syntax for the common pattern of returning an error early if one occurred or otherwise pulling the value out of the `Ok` case and continuing on. The function `bind` returns a `Result`, by putting the `?` after the call, we are saying that if the returned `Result` is the `Err` variant, then just return early with that value.

Otherwise, take the value out of the `Ok` variant and continue to call `workers(8)` on that value. This is a shorthand that makes working with code that might fail much more pleasant. It is functionally equivalent to:

```
let result = HttpServer::new(move || {
  ...
}).bind(("127.0.0.1", self.port));
if result.is_err() {
  return Err(result.err().unwrap());
}
result.unwrap().workers(8).run()
```

Creating our handler

In our code for creating our app we called `service(index)` in our application factory to specify that we have a function called `index` which acts as a service for our app. Let's first decide what we want this handler to do. We are going to look for a particular header in a get request and respond with a message based on the value of that header.

If the header is not present we will respond with a default message. Let's define the structure of this response:

src/lib.rs

```
7  #[derive(Serialize)]
8  struct IndexResponse {
9      message: String,
10 }
```

We create a struct which will be the Rust representation of our response, one field with the name `message` with a `String` value. We then use a special attribute on the struct to derive the `Serialize` trait which we imported earlier from Serde. Let's break down these concepts a bit because they come up a lot in Rust.

Attributes

Attributes are the way of attaching metadata to a variety of things in the language. They can be attached to modules as a whole, structs, functions, and several other constructs. They can attach to the thing they are defined within using the syntax `#![...]` with a `!` after the `#`. For example,

```
fn some_unused_variable() {
  #![allow(unused_variables)]
  let x = ();
}
```

The `allow` attribute is used to turn off a lint warning for the entity that contains the attribute which is the function `some_unused_variable` in this example.

The other format is to attach to the item following the attribute which is what we see with our struct. The `derive` attribute is probably the most common attribute you will encounter. It allows you to implement traits for types without having to do any more work provided the type meets the requirements for the trait to be derived. Most structs will at the very least will derive `Debug` which allows the struct to be printed using the `{:?}` debug format specifier. Any struct can derive `Debug` if its constituents implement the `Debug` trait which all of the builtin types do.

`Debug` is a builtin trait and therefore the attribute to derive it is builtin to the compiler. It is possible to write custom derive logic so that your own traits can be derivable. `Serialize` is just such a custom derivable trait. Now that we have derived `Serialize` any instance of our struct can be serialized by serde into the output format of our choice.

Deriving traits is only one use of the attribute system. There are many other attributes builtin to the compiler for doing things like giving inlining hints and conditional compilation for different architectures.

Now that we have a structure for response data, let's define the handler that will accept requests:

src/lib.rs

```
12 #[get("/")]
13 fn index(req: HttpRequest) -> Result<web::Json<IndexResponse>> {
14     let hello = req
15         .headers()
16         .get("hello")
17         .and_then(|v| v.to_str().ok())
18         .unwrap_or_else(|| "world");
19
20     Ok(web::Json(IndexResponse {
21         message: hello.to_owned(),
22     }))
23 }
```

Actix provides a few attributes for colocating routing information with the handler function. This is how we were able to call `service(index)` without any more information about the type of request or the path. Rocket uses these custom attributes quite heavily for defining the structure of your application. In Actix we can use them like we do here, or we can specify this information in our application factory which we will show in our later handlers. In this case, we define this handler to respond to `GET` requests at the root path.

How handlers work in Rust

Most of the work in defining a handler in all of the Rust web ecosystem is centered around defining the input and output types.

There is quite a bit of work that the libraries, with the help of the compiler, can do for us to extract data from requests and respond with proper HTTP responses while letting us write functions that let us focus on the things that matter.

We will see this in later sections, but idiomatic design using the current web frameworks focuses on the type signature explaining what the function uses. The alternative would be handlers that all take a generic request as input and return generic response as output and then the internals of the function need to be introspected to determine what a handler does.

A set of traits are used for defining how data can be extracted from requests, and for defining how data is to be turned into an HTTP response. A lot of these traits are implemented for us by the library and therefore most of the time we just get to say what types we want to receive and what types we want to respond with and the rest seems like magic. However, under the hood, most of the magic is really just a function of the trait system.

Here we see from the signature that we are returning a `Result` with inner success type of `Json<IndexResponse>`. The `Result` implies that this function can fail, if it does we will return an error, but if it succeeds then the response should be a JSON representation of our `IndexResponse` struct. We also see that we are going to be using something from the `HttpRequest` within the body of our function.

If we took the request as a parameter but did not actually use it then we would get a compiler warning. It is not dramatic so far in this example, but as you build more complex handlers, focusing on the types can take you quite far in understanding an application at a high level.

Given our type signature, let's walk through the body of our handler. Suppose we get a request with the header `hello: Rust`, then we want to respond with the JSON `{"message": "Rust"}`. If we get a request without a `hello` header, then we will respond with `{"message": "world"}`. So the first step is to try to get the value of the `hello` header:

```
req.headers.get("hello")
```

Working with Options

This will return an `Option<&HeaderValue>` where `HeaderValue` is a type that abstracts the bytes that hold the actually data from the request. `Option<T>` is an enum in the standard library with two variants: `Some(T)` and `None`.

The idea of `Option` is to represent the possibility of something not always existing and hence replaces the need for the concept of null found in many other programming languages. The major distinction between null in other languages and `Option` in Rust is that an `Option` is an explicit type that has a `None` variant that you must deal with and thus the concept of null cannot inhabit other types.

In many other languages null can be the value of nearly every type of variable. `Option` is the other main error handling primitive that complements `Result`. Wherein `Result` carries an error value, sometimes you either have something or you don't and in those scenarios `Option` is the more suitable type to use.

We use the function `and_then` defined on `Option` to call a function with the value inside of the option if there is one. In other words, if the header exists, we call our closure with the value, otherwise `and_then` is a no-op on `None`. Our closure has to return an `Option` so that the type in both scenarios matches up. We call `to_str` on the `&HeaderValue` which gives us `Result<&str, ToStrError>` where `ToStrError` is a specific error that explains why the conversion to a `&str` failed.

> Ownership is an important part of Rust and dealing with strings is no different. The most primitive string type is named `str` and is known as a string slice. This is a slice in the same sense that `[i32]` is a slice of signed 32-bit integers. A string slice is a slice of bytes, i.e. it has type `[u8]` and it also is valid Unicode.
>
> The `str` type is almost always encountered as the borrowed variant `&str` which is a reference to a valid Unicode byte array. The reference means that it points to memory owned by someone else. In particular, static string literals are represented with type `&'static str` where the notation `&'static` means a reference to something with a static lifetime. The static lifetime is a special lifetime in Rust which is the entire life of your program. Static strings are compiled into your binary and are therefore owned by the binary. The other type of string has type `String` which is a heap allocated string, i.e. it is a string you own.

As we said before, the closure we pass to `and_then` needs to return an `Option`, but `to_str` returned a `Result`. Luckily `Result` has a handy function called `ok` which takes data from the `Ok` variant of a `Result` and puts it inside the `Some` variant of an `Option`, otherwise it turns the `Err` variant of the `Result` into the `None` variant of `Option` and discards the error. This is exactly the behavior we want here as we are going to treat a malformed header value as the same as a missing header value.

Finally, we want our `hello` variable to just contain a `&str`, but we have an `Option<&str>`. We can again use a helper defined on `Option` called `unwrap_or_else` which will unwrap and return data inside the `Some` variant of the `Option` if it is set,

otherwise in the case of the `None` variant, this function will call the provided function and return the result.

As the first case returns `&str` by extracting the data out of the `Some` variant, the closure we pass to `unwrap_or_else` must also return a `&str`. In this case we just return `"world"` directly as this is our default.

Both `Result` and `Option` have a variety of helper methods defined on them which allow you to write these functional style pipelines which transform data while still dealing with various failure scenarios.

Frequently if you figure out what type of data you want after applying a transformation you can check the type signatures of the methods on the type that you have, be it `Result` or `Option`, to find something that will do what you want. Many of these tools come from the influence of ML style functional programming languages on the design of Rust.

So after all that `hello` contains a `&str` which is either the value of the `hello` header or the literal `"world"`. Let's return it. We need to construct an instance of our `IndexResponse` struct, so the obvious initial attempt might be:

```
IndexResponse {
  message: hello,
}
```

But that will not work. The type of `message` is `String` and the type of `hello` is `&str`. So we need to convert our borrowed string into an owned string so that we can return the data as a response.

Therefore, we need to call some method that explicitly makes this conversion because it is not free. You will find that all of `to_owned()`, `to_string()`, and `into()` would work to do this conversion and you will also see them all in other Rust code. They each use traits which at the end of the day will execute the same code for this use case.

So which one is right? It depends, but in this case because the intent is to turned a borrowed type into the owned variant `to_owned` most accurately describes what is happening. The use of `to_string` can be confusing as we have a string so turning it into a string seems like a weird piece of code. The other option of using `into` which

goes through the `Into` and `From` traits would be a reasonable choice if you care about saving four characters for the sake of being slightly less specialized to this particular scenario.

Once we have our struct that can be serialized to JSON we wrap it in the actix-web `web::Json` type which takes care of doing all the work of properly returning a JSON HTTP response. Finally, we can wrap this response in the `Ok` variant of `Result` to signal that the handler succeeded.

Example requests

example

```
$ curl localhost:8080
{"message":"world"}

$ curl -H "hello: actix" localhost:8080
{"message":"actix"}
```

Summary

We have spent quite a bit of time getting to what is effectively the "Hello, World" of web servers. However, we have learned a lot about Rust to get here. We have learned about the different aggregate data types, structs and enums. We learned about helpful standard library types like `Result` and `Option`, and how to work with them to write clean, robust code. We briefly touched on ownership discussing `String` and `&str`, this will certainly not be our last time talking about ownership. We also discussed implementing traits for types and how this allows us to add behavior to our data.

In the end we have a web server that can respond to data it receives in a header, but we now have a solid foundation of the language that will let us expand our application as we move on. The next step is adding state to this server, along the way we will learn quite a bit more about the language.

Adding State to Our Web App

Recap and overview

We built a basic web server in the previous chapter which is the foundation for this chapter and the next. In this chapter we are going to add some in-memory state to the server. This will require us to explore the concepts of interior mutability, atomic types, as well as other tools which make it safe to share data across threads. The next chapter is focused on state backed by a database which is distinctly different from what we are covering here.

We are going to make the server more complex in this chapter, and sometimes this is exactly what you need to achieve your goals. However you always want to use the right tools for your job, so do not feel like each piece is required for every problem you might be solving.

Adding state

We're going to start with the Actix server we created in the previous chapter, but we are going to make our server slightly more complex by adding state that persists across requests.

First let's update our set of import statements to bring in some new types that we will use for managing state:

src/lib.rs

```
4  use actix_web::{middleware, web, App, HttpServer, Result};
5  use serde::Serialize;
6  use std::cell::Cell;
7  use std::sync::atomic::{AtomicUsize, Ordering};
8  use std::sync::{Arc, Mutex};
```

The `Cell` type we will discuss shortly. The group of things we pull in from `std::sync::atomic` are the tools we need to work with a `usize` that can be modified atomically and therefore is thread-safe. Finally, we pull in `Arc` and `Mutex` from `std::sync` which are tools we will use to safely share and mutate things that are not atomic across multiple threads.

Actix by default will create a number of workers to enable handling concurrent requests. One piece of state we are going to maintain is a unique `usize` for each worker. We will create an atomic `usize` to track this count of workers because it needs to be thread-safe however it only ever needs to increase. This is a good use case for an atomic integer. Let's define a static variable to hold our server counter atomic:

src/lib.rs

```
10 static SERVER_COUNTER: AtomicUsize = AtomicUsize::new(0);
```

Static versus const

There are two things that you will see in Rust code which look similar and which live for the entire lifetime of your program, one is denoted `const` and the other is denoted `static`.

Items marked with `const` are effectively inlined at each site they are used. Therefore references to the same constant do not necessarily point to the same memory address.

On the other hand, `static` items are not inlined, they have a fixed address as there is only one instance for each value. Hence `static` must be used for a shared global variable.

It is possible to have `static mut` variables, but mutable global variables are bad and therefore in order to read/write mutable statics requires the use of the `unsafe` keyword.

Atomics, on the other hand, can be modified in such a way that we do not need to mark the variable as mutable. The `mut` keyword is really a marker for the compiler to guarantee that certain memory safety properties are upheld for which atomics are immune.

Both `static` and `const` variables must have their types given explicitly, so we write the type `AtomicUsize` for our variable. The `new` function on `AtomicUsize` is marked `const` which is what allows it to be called in this static context.

Now we can define the struct which will hold the state for our app:

src/lib.rs

```
12 struct AppState {
13     server_id: usize,
14     request_count: Cell<usize>,
15     messages: Arc<Mutex<Vec<String>>>,
16 }
```

Each worker thread gets its own instance of this state struct. Actix takes an application factory because it will create many instances of the application, and therefore many instances of the state struct. Therefore in order to share information across the different instances we will have to use different mechanisms than we have seen so far.

Defining our state

The first part of the state will be set from the atomic `usize` we declared earlier. We will see how this is set when we get to our updated factory function, but for now we can note that this will just be a normal `usize` that gets set once when this struct is initialized.

The second piece of data will keep track of the number of requests seen by the particular worker that owns this instance of state.

The request count is owned by each worker and changes are not meant to be shared across threads, however we do want to mutate this value within a single request. We cannot just use a normal `usize` variable because we can only get an immutable reference to the state inside a request handler. Rust has a pattern for mutating a piece of data inside a struct which itself is immutable known as interior mutability.

Two special types enable this, `Cell` and `RefCell`. `Cell` implements interior mutability by moving values in and out of a shared memory location. `RefCell` implements interior mutability by using borrow checking at runtime to enforce the constraint that only one mutable reference can be live at any given time.

If one tries to mutably borrow a `RefCell` that is already mutably borrowed the calling thread will panic. As we are dealing with a primitive type as the interior value, namely `usize`, we can take advantage of `Cell` copying the value in and out and avoid the overhead of the extra lock associated with a `RefCell`. `Cell` and `RefCell` are not needed that often in everyday Rust, but they are absolutely necessary in some situations so it is useful to be aware of them.

Finally, the last piece of state is going to be a vector of strings that represent messages shared across all of the workers. We want each worker thread to be able to read and write this state, and we want updates to be shared amongst the workers.

In other words, we want shared mutable state, which is typically where bugs happen. Rust provides us with tools that makes writing safe and correct code involving shared mutable state relatively painless. The state we care about is a vector of strings, so we know we want a `Vec<String>`.

Sharing across threads

We also want to be able to read and write this vector on multiple threads in a way that is safe. We can ensure mutually exclusive access to the vector by creating a `Mutex` that wraps our vector. `Mutex<Vec<String>>` is a type that provides an interface for coordinating access to the inner object (`Vec<String>`) across multiple threads. We will see how this works in the implementation of our handler.

The last piece of the puzzle is that we want to share ownership of this vector. Typically each value in Rust has a single owner, but for this situation we want each thread to be an owner of the data so that the vector lives until the last worker thread exits. The mechanism for this in Rust is to use a reference counted pointer.

There are two variants: `Rc` and `Arc`. They both are generic over a type `T` and provide a reference counted pointer to a value of type `T` allocated on the heap. Calling `clone` on an `Rc` will produce a new pointer to the same value on the heap. When the last `Rc` pointer to a value is destroyed, the pointed-to value will then be destroyed. The A in `Arc` stands for atomic as the reference counting mechanism of an `Rc` is non-atomic and therefore not thread safe.

You cannot share an `Rc` across threads, but you can share an `Arc`. Otherwise they are equivalent. `Rc<T>` uses a trait called `Deref` to allow you to call the methods of `T` directly on a value of type `Rc<T>`. As Rust does not have a garbage collector, it is possible to create memory leaks by creating cycles of reference counted pointers.

There is a non-owning variant called `Weak` which can be used to break such cycles. This is not an issue for us here, but it is important to be aware of especially if you are coming from a garbage collected language.

Okay, now that we have our state defined, let's update our handler to use that state. The first step will be to change our response struct to include information from our state:

src/lib.rs

```
18 #[derive(Serialize)]
19 struct IndexResponse {
20     server_id: usize,
21     request_count: usize,
22     messages: Vec<String>,
23 }
```

We will respond with the id of the server that handled this request, the number of requests this server has seen so far, and the vector of messages.

With our desired response defined, let's change our index handler to do the job:

src/lib.rs

```
#[get("/")]
fn index(state: web::Data<AppState>) -> Result<web::Json<IndexResponse>\
> {
    let request_count = state.request_count.get() + 1;
    state.request_count.set(request_count);
    let ms = state.messages.lock().unwrap();

    Ok(web::Json(IndexResponse {
        server_id: state.server_id,
        request_count,
        messages: ms.clone(),
    }))
}
```

Extracting data from requests

We have updated the signature of our function to take the state as input while still returning a JSON representation of our response struct. It may seem a bit magical to just define the input parameter of our handler to be the state rather than having to figure out how to get that from our server or the request. The mechanism that allows this is a trait called `FromRequest` and the generic term for this concept is an extractor.

Extractors are types that implement the `FromRequest` trait which allow types to define how they are constructed from a request. The underlying framework provides many extractors for things like query parameters, form input, and in this case getting the state that the application factory created for the current worker.

This turns out to be a powerful and safe abstraction because the compiler is able to provide a lot of guarantees about what data is and is not available. Any type that implements `FromRequest` can technically fail to extract said type and thus uses `Result` in the implementation. You can define your handler to take a `Result<T>` or an `Option<T>` for any `T` that implements `FromRequest` to be able to handle the failure of extraction in your handler.

We will see below how the `AppState` gets constructed in the application factory, but for now we can assume that our handler will be called with an instance of `web::Data<AppState>` which is just a wrapper around our state that handles the `FromRequest` implementation. This wrapper implements the `Deref` trait so that we can treat a value of `Data<AppState>` effectively as if it was a value of `AppState`.

The first thing we do in our handler is to update the number of requests this server has handled. As the `request_count` field is a `Cell`, we have to use the `get` and `set` methods on `Cell` to update the value. We could have used an atomic `usize` for this variable as well, but we chose to use a `Cell` to demonstrate how to use it.

The reason that we cannot mutate `request_count` directly is that our `state` variable is an immutable reference. There is no way for us to update `server_id` for example. Hence, we first use `get` to get the current value inside our cell and then add one to account for the current request. We use `set` to store this new value back into the `Cell`.

Effectively working with locks

Next we need to get access to our messages vector. Recall that `state.messages` has type `Arc<Mutex<Vec<String>>>`. First `Arc` implements `Deref` so that when we call a method on `state.messages` it will automatically get called on the value of type `Mutex<Vec<String>>`. To get access to the data inside the mutex we call the `lock` method on the mutex.

The `lock` method blocks until the underlying operating system mutex is not held by another thread. This method returns a `Result` wrapped around a `MutexGuard` which is wrapped around our data. The `Result` that is returned will be an error only if the mutex is poisoned which basically means a thread paniced while holding the lock and likely your program is in a bad state. We choose to use the `unwrap` method on `Result` which pulls data out of the `Ok` variant, but instead panics on the `Err` variant. We do this because if we get back an error from `lock` we don't really know how to handle that state of the world so also panicing is not a bad option. This might not always be the right choice, but often you will see `lock().unwrap()` used with mutexes.

The type of the variable we get from `state.messages.lock().unwrap()` is actually a `MutexGuard<Vec<String>>`. Again through the magic of `Deref` you can just treat this

as the vector of strings we want.

It is relevant to us because it explains how mutexes work in Rust. RAII (Resource Acquisitions Is Initialization) is a pattern for managing resources which is central to Rust. In particular, when a value goes out of scope, a special method called `drop` is called by the compiler if the type of the value implements the `Drop` trait. For a `MutexGuard`, the mutex is locked when the guard is constructed and the lock is unlocked in the guard's `drop` method.

Therefore, the lock is only locked for as long as you have access to the guard. Additionally, you only have access to the data protected by the mutex through the guard. Hence, the data is only accessible while the lock is locked. You don't have to worry about calling unlock at the right time or ensuring that you actually locked the mutex in all the places that you read or write the vector of messages. All of that is taken care of for you by the compiler.

Responding with data

Finally, we can construct an instance of our response struct to be serialized and returned as JSON. The one piece to note is that we call `ms.clone()` for the value we set on the `messages` field of our response struct. The `clone` method creates an explicit copy of a value if the type implements the `Clone` trait.

We cannot just pass the messages vector directly because that would move ownership and that is not what we want to do (nor even possible because it is shared). We want to return a copy of the vector of messages to be serialized. Because this copying might be expensive Rust does not do it implicitly, rather you are required to state that you want it to happen explicitly by calling `clone`.

For things that can be copied cheaply, there is a separate trait called `Copy` which will result in implicit copies being created. You can decide what behavior you want for the types you create based on what traits you derive or implement.

Now that our handler is in place, we just need to update the application factory inside our `run` method to incorporate our state:

src/lib.rs

```
47      pub fn run(&self) -> std::io::Result<()> {
48          let messages = Arc::new(Mutex::new(vec![]));
49          println!("Starting http server: 127.0.0.1:{}", self.port);
50          HttpServer::new(move || {
51              App::new()
52                  .data(AppState {
53                      server_id: SERVER_COUNTER.fetch_add(1, Ordering::Se\
54  qCst),
55                      request_count: Cell::new(0),
56                      messages: messages.clone(),
57                  })
58                  .wrap(middleware::Logger::default())
59                  .service(index)
60          })
61          .bind(("127.0.0.1", self.port))?
62          .workers(8)
63          .run()
64      }
```

We create the shared messages vector outside of the application factory closure with the line:

```
let messages = Arc::new(Mutex::new(vec![]));
```

We do this so that each worker can actually share the same messages array rather than each of them creating their own vector which would be unconnected from the other workers.

To add state to the application we use the `data` method on `App` and pass in the initial value of our state. This is what makes the `web::Data<AppState>` extractor work.

Constructing our state

We have three parts to our state that each get setup differently. First we have the id of the worker which we call `server_id`. When each of the workers start their instance of the app, this `AppState` struct will be constructed.

For `server_id`, we use the global `SERVER_COUNTER` variable to set the value for the server id. The `fetch_add` method will atomically add an amount equal to the first argument to the global variable and return whatever value was in the variable prior to the addition. So the first time this is called `SERVER_COUNTER` will be incremented from zero to one, but the `server_id` will be set to zero as that was the value in `SERVER_COUNTER` before `fetch_add` was called.

The second argument to `fetch_add` controls how atomic operations synchronize memory across threads. The strongest ordering is `SeqCst` which stands for sequentially consistent. The best advice is to use `SeqCst` until you profile your code, find out that this is a hot spot, and then can prove that you are able to use one of the weaker orderings based on your access pattern.

The second piece of data is the `request_count`. We initialize this value by putting zero inside a `Cell`. Each thread will own its own `Cell` so we just construct the cell inside the application factory closure which is executed by the worker thread and therefore has affinity to that thread which is what we desire.

Finally, we clone the `Arc` value that wraps the shared messages value which means that we create a new pointer to the shared data. We have to use `Arc` instead of `Rc` because all of the `clone` calls are happening on different threads. If you tried to use an `Rc` here the compiler would give you an error because it can tell you are moving across a thread boundary.

The rest of our run function is exactly the same. In particular, the `service(index)` call is the same even though we changed the signature of the handler. This is due to some generic programming done by the underlying actix library to deal with handlers with a flexible number of extractor input arguments.

Example requests

example

```
$ curl localhost:8080
{"server_id":0,"request_count":1,"messages":[]}

$ curl localhost:8080
{"server_id":1,"request_count":1,"messages":[]}

$ curl localhost:8080
{"server_id":2,"request_count":1,"messages":[]}

$ curl localhost:8080
{"server_id":3,"request_count":1,"messages":[]}
```

Receiving input

Our app can now respond with a list of messages stored on the server. However, this is of limited use without a way to actually add messages. Let's add the ability to post messages which will get added to our list of messages.

First we want to be able to accept JSON data as input, so we will import the `Deserialize` item from `serde` so that we can derive the ability to construct structs from JSON data:

src/lib.rs

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

Let's define our input and output data for the method that accepts data:

src/lib.rs

```
#[derive(Deserialize)]
struct PostInput {
    message: String,
}

#[derive(Serialize)]
struct PostResponse {
    server_id: usize,
    request_count: usize,
    message: String,
}
```

Our input will just be of the form `{"message": "some data"}`, so we create a struct `PostInput` and derive `Deserialize`. We will then be able to use Serde to turn JSON data with that format into instances of our struct. Serde has defaults that usually do what you want, but if you wanted to change the name of a field or otherwise customize the input/output format you can use custom attributes on your struct.

The output will be very similar to our `IndexResponse` in that we will include information about the worker that handled this request, but instead of returning the whole list of messages, we will just echo back the message that was input.

Given our input and output data format, let's define our handler:

src/lib.rs

```
fn post(msg: web::Json<PostInput>, state: web::Data<AppState>) -> Resul\
t<web::Json<PostResponse>> {
    let request_count = state.request_count.get() + 1;
    state.request_count.set(request_count);
    let mut ms = state.messages.lock().unwrap();
    ms.push(msg.message.clone());

    Ok(web::Json(PostResponse {
        server_id: state.server_id,
        request_count,
```

```
60          message: msg.message.clone(),
61      }))
62  }
```

This handler will handle our posted messages, we chose the name `post` for the name of the function, but it has nothing to do with the HTTP method used for taking data.

Note that we are not using an attribute to define the method and route as we will construct this service differently in our `run` method to demonstrate the other style of route configuration.

Looking at our handler's function signature we see that we expect JSON input data, we will access the state of our app, and we are returning a JSON representation of our `PostResponse` struct. The `web::Json<T>` extractor attempts to deserialize the request body using serde into the type specified. If deserialization fails for any reason, an error is returned. This behavior can be customized as we will see later.

Note that we can take in as many request extractors as we need to make our handler do what we want. Technically the library implements the ability to take up to nine extractors, but you can take an arbitrary number by combining them into a tuple rather than having them be separate arguments.

Thanks to the extractor doing the hard work of getting the JSON from the request and constructing an instance of `PostInput`, we know that inside our method we have input data in the format we expect.

Therefore, we can do something similar to our index method except we want to modify our messages vector. We lock the mutex to get access to the vector and then we push this message onto the end. Note that we declare our message vector variable `ms` to be mutable so that we can call `push` which is safe because as we are holding the mutex we have exclusive read/write access to the vector.

We have to clone the message as we push it into the vector because this vector owns each element and we only have a borrowed reference to our `PostInput` data. This also explains why we clone the message when we return it as part of our `PostResponse`.

We can now update our application factory to define the route leading to our `post` handler:

src/lib.rs

```
96                  .wrap(middleware::Logger::default())
97                  .service(index)
98                  .service(
99                      web::resource("/send")
100                         .data(web::JsonConfig::default().limit(4096))
101                         .route(web::post().to(post)),
102                 )
```

The `service` method takes a resource definition. For our `index` method because we used the special attribute syntax this resource definition is generated for us. For the `post` handler, we are going to create our own resource. The code is pretty self explanatory.

First we create a resource for the specific path `/send` with `web::resource("/send")`.

The `data` method is used for specifying route specific data or for configuring route specific extractors. We use it here to demonstrate how to configure the JSON extractor for this route by setting a limit on the number of bytes to deserialize to 4096 bytes.

We then declare the route configuration for this resource by passing route data to the `route` method. There are methods for each HTTP method, so we use `web::post()` to say that this route requires a `POST` request.

Finally, `to` is called with our handler function `post` to indicate which function to call for this route.

While we are at it, let's also create a post request that clears out all the messages:

src/lib.rs

```
63 #[post("/clear")]
64 fn clear(state: web::Data<AppState>) -> Result<web::Json<IndexResponse>\
65 > {
66     let request_count = state.request_count.get() + 1;
67     state.request_count.set(request_count);
68     let mut ms = state.messages.lock().unwrap();
69     ms.clear();
70
71     Ok(web::Json(IndexResponse {
72         server_id: state.server_id,
73         request_count,
74         messages: vec![],
75     }))
76 }
```

This is similar to our index request so we will just repurpose the response type and also return `IndexResponse` from `clear`. The implementation follows our `post` handler, except instead of pushing a new message onto our vector, we mutate it by calling `clear` to remove all messages.

Lastly we just return an empty vector in our response because we know that we just cleared out our vector.

As we used the attribute syntax to define the route as a post request to `/clear`, we need only add a service to our app:

src/lib.rs

```
102                 )
103                 .service(clear)
104         })
105         .bind(("127.0.0.1", self.port))?
106         .workers(8)
```

Example requests

example

```
$ curl localhost:8080
{"server_id":0,"request_count":1,"messages":[]}

$ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
lo"}' localhost:8080/send
{"server_id":1,"request_count":1,"message":"hello"}

$ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
lo again"}' localhost:8080/send
{"server_id":2,"request_count":1,"message":"hello again"}

$ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
lo"}' localhost:8080/send
{"server_id":3,"request_count":1,"message":"hello"}

$ curl localhost:8080
{"server_id":4,"request_count":1,"messages":["hello","hello again","hel\
lo"]}

$ curl -X POST localhost:8080/clear
{"server_id":5,"request_count":1,"messages":[]}

$ curl localhost:8080
{"server_id":6,"request_count":1,"messages":[]}

$ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
lo after clear"}' localhost:8080/send
{"server_id":7,"request_count":1,"message":"hello after clear"}

$ curl localhost:8080
{"server_id":0,"request_count":2,"messages":["hello after clear"]}
```

Custom error handling

We can now post messages to our app and get them back out. But currently if you made a mistake when posting a message and sent data like `{"my_message": "hello"}`, the app would return a 500. This is not a great user experience. There are a variety of mechanisms for dealing with errors but we are going to focus on this one particular case where we want to custom the error handler associated with the JSON decoding step.

First, let's add bring in some imports from `actix_web` that will make dealing with errors less verbose:

src/lib.rs

```
4  use actix_web::{
5      error::{Error, InternalError, JsonPayloadError},
6      middleware, web, App, HttpRequest, HttpResponse, HttpServer, Result,
7  };
```

Rather than returning a 500 response with no data, we are going to respond with a more appropriate status code along with an error message. As the rest of our functions are returning JSON, we want our error handler to also return JSON. Therefore, let's define a struct to represent the response from our error handler:

src/lib.rs

```
41 #[derive(Serialize)]
42 struct PostError {
43     server_id: usize,
44     request_count: usize,
45     error: String,
46 }
```

This is the same structure as our other responses, but we are going to have an error field with a string that holds a message about what went wrong.

Given the data we want to output, let's define the handler function that will be called in the case of a JSON error:

src/lib.rs

```
88 fn post_error(err: JsonPayloadError, req: &HttpRequest) -> Error {
89     let extns = req.extensions();
90     let state = extns.get::<web::Data<AppState>>().unwrap();
91     let request_count = state.request_count.get() + 1;
92     state.request_count.set(request_count);
93     let post_error = PostError {
94         server_id: state.server_id,
95         request_count,
96         error: format!("{}", err),
97     };
98     InternalError::from_response(err, HttpResponse::BadRequest().json(p\
99 ost_error)).into()
100 }
```

The type for this handler is defined by the `JsonConfig` type and is not as flexible as the other handlers that we can define. In particular, the signature of the function has to be exactly this, we cannot use extractors to get different input, and we have to return the `Error` type from `actix_web`.

This might change in a future version of the library, but for now this is the required.

The first argument is an instance of the enum `JsonPayloadError` which contains information about what went wrong. The second argument is a reference to the request that led to this error. We still want to access our state because we want to update the request count and get the id of the current worker to return in our response.

We can do this by doing what the state extractor would have done for us and pull our state out of the request's extensions.

Generic return types

Actix uses a type safe bag of additional data attached to requests called extensions. The state is just the value inside of the extensions with type `web::Data<AppState>`.

We get a reference to the extensions by calling the `extensions` method on the request. The line to pull out the state looks a bit complicated but we can break it down. The extensions have a generic function called `get` which has the signature:

```
fn get<T>(&self) -> Option<&T>
```

This is a function that is returns a reference to a type that was previously stored as an extension. It is up to the caller (i.e. us) to say what type we want to get back.

We have to give the compiler some help by putting a type annotation somewhere so that `get` knows what type we want. We do that by using the `::<>` turbofish syntax, namely `get::<web::Data<AppState>>` means call `get<T>()` with `T` bound to the type `web::Data<AppState>`. The implementation has to know how to handle any generic type but for us we just know that we will get back the data we want if it was previously stored, otherwise the `Option` will be the `None` variant.

> The actual signature of `get` is `fn get<T: 'static>(&self) -> Option<&T>` but the `'static` lifetime bound on the generic type `T` is not important for the current discussion.

For the sack of brevity we call `unwrap` on the `Option` we get back to get direct access to our state. We know that if our app is properly configured then we will always get back our state and thus this `Option` should never be `None`.

However, it is worth noting that if we did not configure the state properly we would not see this error until runtime when this call to `unwrap` panics. We could match on the value of the option and handle the `None` case by returning a fallback error, but that is a future improvement.

Creating useful errors

Once we have the state, we can work with it in the exact same way that we did in our other handlers where the extractor took care of the previous step for us.

We create our `PostError` struct with the id and request count gathered from the state. We set the error message to be the string representation of the `JsonPayloadError` that

was passed to us. The `format` macro takes a format string along with the necessary variables to fill in the placeholders and returns an owned `String`. In this case, we use the format string `"{}"` to use the `Display` trait implementation of your error to get a nice message.

This is in contrast to `"{:?}"` which uses the `Debug` trait implementation which would not be a nice message. In a real app, you would probably want a more user friendly message than even just displaying this error. One approach would be to match on the different variants of the `JsonPayloadError` enum and make our own message in the different scenarios.

Finally, we want to construct an `Error` with our struct turned into JSON. `InternalError` is a helper provided by actix to wrap any error and turn it into a custom response. So we call the constructor `from_response`, passing in the `JsonPayloadError` which gets stored as the cause of the error, and then the second argument is the custom response we want to return.

The `HttpResponse` struct has a variety of helpers for building responses, one of which is `BadRequest` which sets the status code to 400 which by the spec means the server is working properly but your request was bad for some reason. This method returns a response builder which has a `json` method that can take anything that is serializable into JSON and sets it as the response body.

Interpreting compiler errors

That covers the entirety of the final line in our handler except for that call to `into()` at the end. If you leave off this call you will get a compiler error that ends with:

```
= note: expected type `actix_web::Error`
           found type `actix_web::error::InternalError<actix_web::error\
::JsonPayloadError>`
```

This is correct, we are returning an `InternalError` struct when our function claims to return an `Error` struct. So we can't just return that struct directly. However, Rust has a pair of traits `std::convert::From` and `std::convert::Into` which define explicit transformations between types. The documentation states that `From<T>` for

`U` implies `Into<U>` for `T` which means that typically only one side of the conversion is implemented.

In particular, libraries usually implement the `From` trait to define how to construct an instance of their type from other things. Users can then create their own types and call `into` to hook into the conversion facilities provided by the library.

To make this more concrete, in this case, `actix_web::Error` implements `From<T>` for any `T` which implements the `ResponseError` trait. As `InternalError` implements the `ReponseError` trait, we can explicitly say that we want to convert our `InternalError` into an `Error` and let the `From` implementation of `Error` take care of what that means.

In order to use our new error handler, we simply declare that we want to use it by calling the `error_handler` method on the `JsonConfig` setup in our application factory:

src/lib.rs

```
121                 .service(index)
122                 .service(
123                     web::resource("/send")
124                         .data(
125                             web::JsonConfig::default()
126                                 .limit(4096)
127                                 .error_handler(post_error),
128                         )
129                         .route(web::post().to(post)),
130                 )
```

While we are here let's also update the format of the logger middleware. Near the top of our file let's define a string constant that specifies our desired log format:

src/lib.rs

```
13 const LOG_FORMAT: &'static str = r#""%r" %s %b "%{User-Agent}i" %D"#;
```

The details for what is available can be found in the `actix_web` documentation. This format is similar to the default but removes a few of the less useful pieces of information, and we change the reporting of the time spent processing the request to

be in milliseconds by using `%D`. We are using a raw string because we want to include quotation marks inside our string without having to escape them.

The syntax is the character `r` followed by zero or more `#` characters followed by an opening `"` character. To terminate the string you use a closing `"` character followed by the same number of `#` characters you used at the beginning. Here we use one `#` character, but there are cases where you want to use more, for example if you want to embed `"#` inside your string.

We then change our middleware specification in our application factory to use this format. We do this by replacing `middleware::Logger::default()` with:

src/lib.rs

```
120                     .wrap(middleware::Logger::new(LOG_FORMAT))
```

Example requests

example

```
1  $ curl -X POST -H "Content-Type: application/json" -d '{"bad": "hello"}\
2  ' localhost:8080/send
3  {"server_id":0,"request_count":1,"error":"Json deserialize error: missi\
4  ng field `message` at line 1 column 16"}
5
6  $ curl localhost:8080
7  {"server_id":1,"request_count":1,"messages":[]}
8
9  $ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
10 lo"}' localhost:8080/send
11 {"server_id":2,"request_count":1,"message":"hello"}
12
13 $ curl localhost:8080
14 {"server_id":3,"request_count":1,"messages":["hello"]}
```

Handling path variables

Before we wrap up, let's add one more route to our API to show the basics of handling variable paths. Rather than having to get the entire list of messages, let's add a GET request to `/lookup/{index}` which will attempt to get only the message at that particular index in our list. First we will define a struct to represent this response:

src/lib.rs

```
#[derive(Serialize)]
struct LookupResponse {
    server_id: usize,
    request_count: usize,
    result: Option<String>,
}
```

The different part of this struct from our previous responses is the `result` field which has type `Option<String>`. We use an `Option` because the lookup might fail if the index happens to be out of bounds of the current vector of messages. The `None` variant will be serialized to `null` in JSON, and the `Some` variant will serialize to just the inner data.

With our desired response defined, let's create the handler for this lookup operation:

src/lib.rs

```
#[get("/lookup/{index}")]
fn lookup(state: web::Data<AppState>, idx: web::Path<usize>) -> Result<\
web::Json<LookupResponse>> {
    let request_count = state.request_count.get() + 1;
    state.request_count.set(request_count);
    let ms = state.messages.lock().unwrap();
    let result = ms.get(idx.into_inner()).cloned();
    Ok(web::Json(LookupResponse {
        server_id: state.server_id,
        request_count,
        result,
```

```
119     }))
120 }
```

We use an attribute to define the route for this handler as a GET request to `/lookup/{index}`. This syntax says to match the exact path `/lookup/` and then expect a variable afterwards. The name inside the braces in the attribute defining the route is not actually important for our use case, but can be good documentation.

The signature of our input types has changed to include a `web::Path` extractor in addition to the `Data` extractor we use again because we still want to work with the state.

The `Path` extractor uses the generic type specified, in this case `usize`, to attempt to deserialize the path segment to this type. If we had multiple path segments, then we would pass a tuple with the different expected types in order to allow for deserialization. You can also pass a custom type that implements Deserialize to handle more complex use cases.

Our implementation of `lookup` is quite similar to handlers we have seen before, except for:

```
let result = ms.get(idx.into_inner()).cloned();
```

First we call `into_inner` on the `Path<usize>` variable. This converts the `Path` wrapper into the inner type it is wrapping, in this case a `usize`. Then, we use that `usize` variable as the argument to the `get` method on our vector.

The `get` method on `Vec<T>` returns `Option<&T>`, that is it maybe returns a reference to one of the elements inside the vector. This method does not modify the vector, so it cannot return `Option<T>` as that would require moving data out of the vector and therefore modifying the vector. As the variable `ms` was not declared `mut` we know that we are only ever calling methods which do not mutate the vector.

An `Option` is returned because the index passed in might be out of bounds so this is a safe way of accessing data in the vector without having to manually check that the index is valid.

Our result variable needs to be of type `Option<String>`, but we have a variable of type `Option<&String>`. The `Option` enum implements a method `cloned` which

converts `Option<&T>` to `Option<T>` by cloning the inner data in the `Some` case, and doing nothing in the `None` case. This is exactly the behavior we want.

The last piece of the puzzle is to hook the `lookup` up to our app by declaring it as a service in our application factory:

src/lib.rs

```
150                 )
151                 .service(clear)
152                 .service(lookup)
153         })
154         .bind(("127.0.0.1", self.port))?
```

Example requests

example

```
1  $ curl localhost:8080
2  {"server_id":0,"request_count":1,"messages":[]}
3
4  $ curl localhost:8080/lookup/2
5  {"server_id":1,"request_count":1,"result":null}
6
7  $ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
8  lo"}' localhost:8080/send
9  {"server_id":2,"request_count":1,"message":"hello"}
10
11 $ curl -X POST -H "Content-Type: application/json" -d '{"message": "hel\
12 lo again"}' localhost:8080/send
13 {"server_id":3,"request_count":1,"message":"hello again"}
14
15 $ curl -X POST -H "Content-Type: application/json" -d '{"message": "goo\
16 dbye"}' localhost:8080/send
17 {"server_id":4,"request_count":1,"message":"goodbye"}
18
19 $ curl localhost:8080
```

```
{"server_id":5,"request_count":1,"messages":["hello","hello again","goo\
dbye"]}

$ curl localhost:8080/lookup/2
{"server_id":6,"request_count":1,"result":"goodbye"}

$ curl localhost:8080/lookup/0
{"server_id":7,"request_count":1,"result":"hello"}

$ curl localhost:8080/lookup/foo
-> 404

$ curl localhost:8080/lookup/99
{"server_id":1,"request_count":2,"result":null}
```

Wrapping up

We added some realistic pieces to our web server: state shared across worker threads, handling structured input data, and variable URL paths. Along the way we learned about interior mutability, sharing ownership via reference counting, and safely sharing data across threads with locks.

Working effectively with multiple threads is often thought of as a dangerous activity. Such an attitude is fully justified in languages where the compiler is not helping you get things correct. This is one reason many languages are built to hide this complexity. Hopefully, after working with locks to share data in this chapter you feel like Rust's approach is not as hard.

Even More Web

Crates to know

The vibrant Rust ecosystem makes building complex applications in Rust much easier. This ecosystem continues to grow so that even if something is currently missing, chances a new crate is on the horizon that will solve your problem. And if not, you can help the community by writing a crate yourself.

We will expand our understanding of building a web application by adding persistence via a SQL database. There are a few tools for working with SQL in Rust, from directly accessing the database via wrappers around C libraries to full blown Object Relational Mapping (ORM) libraries. We will take somewhat of a middle-ground approach and use the Diesel[36] crate.

Diesel

Diesel is both an ORM and a query builder with a focus on compile time safety, performance, and productivity. It has quickly becoming the standard tool for interacting with databases in Rust.

Rust has a powerful type system which can be used to provide guarantees that rule out a wide variety of errors that would otherwise occur at runtime. Diesel has built abstractions that eliminate incorrect database interactions by using the type system to encode information about what is in the database and what your queries represent. Moreover, these abstractions are zero-cost in the common cases which allows Diesel to provide this safety with the same or better performance than C.

The crate currently supports three backends: PostgreSQL, MySQL, and Sqlite. Switching between databases is not completely free as certain features are not supported by all backends. However, the primary interaction between your code and the database

[36] https://diesel.rs

is in Rust rather than SQL, so much of the interactions with Diesel are database agnostic. For managing common database administration tasks like migrations, Diesel provides a command line interface (CLI) which we will show how to use.

The Diesel getting started[37] is a great resource for an overview of how Diesel works and what it can do.

Building a blog

We are going to build a JSON API around a database that represents a blog. In order to capture most of the complexity of working with a database we will have a few models with some relationships. Namely, our models will be:

- Users
- Posts
- Comments

A Post will have one User as an author. Posts can have many Comments where each Comment also has a User as author. This provides enough opportunity for demonstrating database interactions without getting overwhelmed by too many details.

We will start out by getting all of the necessary infrastructure in place to support Users. This will involve putting a few abstractions in place that are overkill for a single model, however they will pay dividends when we subsequently add Posts and Comments.

As we have already gone through quite a bit of details related to Actix and building an API, the focus here will be more on the new parts related to working with persistence. Therefore some of the details of working with `actix-web` will be assumed.

Getting setup

Let's get started like with all Rust projects and have Cargo generate a new project:

[37] http://diesel.rs/guides/getting-started/

```
$ cargo new blog-actix
```

Our first step will be editing our manifest to specify the dependencies that we are going to need:

Cargo.toml

```
[package]
name = "blog-actix"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
actix-web = "1.0"

env_logger = "0.6"
futures = "0.1"
serde = "1.0"
serde_json = "1.0"
serde_derive = "1.0"

diesel = { version = "^1.1.0", features = ["sqlite", "r2d2"] }
dotenv = "0.10"
```

The new dependencies beyond what we have previously used with `actix-web` are `diesel`, and `dotenv`. The `diesel` dependency we have already discussed, but there is a bit of a new twist here.

Cargo supports the concept of features which allow crates to specify groups of functionality that you can select when you depend on a crate. This is typically used for conditionally including transitive dependencies and conditional compilation to either include or exclude code based on what you do or don't need. Good crates allow you to pick and choose only what you want to minimize compilation times and binary sizes. The Rust compiler can do some work to remove code that is unused in your final binary but using features is one way to ensure this happens and make it explicitly what you are using.

One thing Diesel uses features for is to specify what backend you want to use. For our purposes we are going to use Sqlite. As an embedded file based database we will be able to work with persistence without having to setup the external dependency of a database server. We will be clear as to what parts of this code depend on this database choice.

The other feature of Diesel that we are specifying, `r2d2`, adds the `r2d2` generic database connection pool crate as a dependency of Diesel and turns on some functionality. Any reasonable production system will use a connection pool for interacting with a database, the reasons are best described in the `r2d2` documentation:

> Opening a new database connection every time one is needed is both inefficient and can lead to resource exhaustion under high traffic conditions. A connection pool maintains a set of open connections to a database, handing them out for repeated use.

Finally, we include `dotenv` as a dependency which is a tool for managing environment variables. By default Dotenv looks for a file named `.env` in the current directory which lists environment variables to load. As we will need it later, let's create this file with one variable `DATABASE_URL` with a file URL to a file in the current directory which will hold our Sqlite database:

<<.env[38]

Installing the Diesel CLI

As we previously mentioned, Diesel has a CLI for managing common database tasks. Cargo has the ability to install binary crates on your system via the `cargo install` command. Therefore, we can install the Diesel CLI with:

```
$ cargo install diesel_cli --no-default-features --features sqlite
```

By default this installs a binary at `~/.cargo/bin` but it is possible to configure this.

As we mentioned Diesel uses features for turning on and off certain functionality. Crates that use features typically have a default set that is turned on if you otherwise

[38] code/intermediate/blog-actix/.env

do not specify anything. It is possible to turn off this default behavior via the command line argument `--no-default-features`, and for the CLI we do this because the default is to include support for all three database backends. This will cause errors running CLI commands if you do not have some of the necessary components installed. So we turn off the default and then turn on only Sqlite support via `--features sqlite`.

Migrations

The Diesel CLI binary is named `diesel`, so we can setup our project for working with Diesel by running the setup command:

```
diesel setup
```

By default this assumes the existence of an environment variable named `DATABASE_URL` or a `.env` file with this variable defined. It is possible to pass this manual to each CLI command, but using one of the aforementioned methods is much more convenient. This is one reason we created the `.env` file above.

This will create a migrations directory as well as a `diesel.toml` file. If you are using Postgres this command will also create a migration that creates a SQL function for working with timestamps. This does not happen for other backends.

Diesel manages migrations using a directory called `migrations` with a subdirectory for each migration. The name of the subdirectories are a timestamp prefix followed by a name. Within each migration directory are two self-explanatory files: `up.sql` and `down.sql`. Diesel uses SQL for migrations rather than a custom DSL. Therefore changing databases requires rewriting most of your migrations.

Running migrations

The primary use for the CLI is managing migrations which uses the `migration` command with further subcommands. To see all migrations and whether they have been applied we use the `list` subcommand:

```
diesel migration list
```

To run all pending migrations we use the `run` subcommand:

```
diesel migration run
```

You can get help for `diesel` in general by calling `diesel --help` or for a particular command by passing `--help` with that command, i.e.:

```
diesel migration --help
```

Schema

So you manage your database using the Diesel CLI and a set of migration files written in SQL. But all of that operates outside of Rust. To connect your database to Rust, Diesel uses a schema file that is a code representation of your database. Running the migrations also modifies this code representation at `src/schema.rs`. The name of this file and whether this happens can be controlled via settings in `diesel.toml`, but the default is usually what you want.

Users

Let's get started creating our application which will support managing users. We are not going to get into the weeds of authentication or authorization, rather our focus will be on manipulating persisted data via a JSON API.

Create users migration

The first step is to add a migration that will create the database table `users` to hold our users:

```
diesel migration generate create_users
```

This creates a directory `migrations/YYYY-MM-DD-HHMMSS_create_users` with two empty files. In `up.sql` let's put the SQL for creating our `users` table:

migrations/2019-04-30-025732_create_users/up.sql

```
CREATE TABLE users (
  id INTEGER PRIMARY KEY NOT NULL,
  username VARCHAR NOT NULL
)
```

Each user has an `id` which will be the primary key for fetching users as well as the key used for referencing users in other tables. We also require each user to have a `username` which is a string. You can get arbitrarily creative here depending on your domain, but for simplicity we only have these two columns.

This syntax is specific to the Sqlite backend so this it should be clear why all migrations could need to be rewritten if you decide to use a different backend. For example, some databases allow you restrict the size of `VARCHAR` columns which might be a reasonable thing to do for a username, but Sqlite does not actually enforce any limit you happen to write.

The corresponding `down.sql` file should perform whatever transformations are necessary to undue what happens in `up.sql`. In this case as the up migration is creating a table, we can drop the table in our down migration:

migrations/2019-04-30-025732_create_users/down.sql

```
DROP TABLE users
```

You can do whatever you want in `up` and `down`, but for your own sanity, the schema should be the same before running the migration and after running up followed by down. That is down should revert the schema to the prior state. As some migrations will update data in the database it is not necessarily true that the data in the database is preserved by running up followed by down. The reversibility of migrations is typically only a statement about the schema, but the exact semantics are up to you.

Make username unique

We create yet another migration, this time to add an index to our users table. We do this to ensure that usernames are unique in our system and that we can lookup users by their username quickly. First we have diesel create the files for us with:

```
diesel migration generate index_username
```

Then we add the code to create the index to `up.sql`:

migrations/2019-04-30-025922_index_username/up.sql

```
1 CREATE UNIQUE INDEX username_unique_idx ON users (username)
```

Again this is Sqlite syntax, although all backends have a similar syntax for this operation. The important part of this index is the `UNIQUE` keyword. This let's us rely on the database for the enforcement of unique usernames rather than introducing racy code that tries to manage this at the application layer.

As before, we want our down migration to reverse what we did in up, so we drop the index in `down.sql`:

migrations/2019-04-30-025922_index_username/down.sql

```
1 DROP INDEX username_unique_idx
```

Schema

We run our migrations via the Diesel CLI with:

```
1 diesel migration run
```

Once this runs successfully two things will be true. First, the database file at `blog.db` will be in the state after running all of our up migrations. You can verify this by opening the Sqlite shell:

```
sqlite3 blog.db
```

and dumping the schema:

```
sqlite> .schema
CREATE TABLE __diesel_schema_migrations (
  version VARCHAR(50) PRIMARY KEY NOT NULL,
  run_on TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
CREATE TABLE users (
  id INTEGER PRIMARY KEY NOT NULL,
  username VARCHAR NOT NULL
);
CREATE UNIQUE INDEX username_unique_idx ON users (username);
```

Note that the `__diesel_schema_migrations` table is automatically created by Diesel and it is how the CLI knows which migrations have or have not run.

The second thing that happens is the file `src/schema.rs` is updated with Rust code which Diesel uses to understand the state of your database. This file should look like:

src/schema.rs

```
1 table! {
2     users (id) {
3         id -> Integer,
4         username -> Text,
5     }
6 }
```

It doesn't look like much because of the magic of the macros that Diesel provides. Here only the `table!` macro is used, but there are a few more that we will encounter as our data model evolves.

Building the application

With the database taken care of for the moment, we turn now to our actual application. We are going to build out the scaffolding which supports users and also will be easily extensible later on.

Main

As we have done before, we are going to split our application into a small `main.rs`, which is the binary entry point, and keep everything else in a library which our main can call in to. So without further ado, let's add the following to `main.rs`:

src/main.rs

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

fn main() -> std::io::Result<()> {
    dotenv().ok();

    env::set_var("RUST_LOG", "actix_web=info");
    env_logger::init();
    let database_url = env::var("DATABASE_URL").expect("DATABASE_URL mu\
st be set");
    let app = blog_actix::Blog::new(8998);
    app.run(database_url)
}
```

Everything here we have seen in our simpler web applications except the interaction with `dotenv`. Calling `dotenv().ok()` sets environment variables based on the contents of the `.env` file in the current directory and ignores any error that might result. Dotenv only sets environment variables from that file if they are not already set so you can always override the file by setting the variable directly in your environment before running the program.

As we have a `.env` file with `DATABASE_URL` defined we will end up with that environment variable set so our call to `env::var("DATABASE_URL")` will succeed. We pass this URL to the `run` call of our app which kicks off everything.

Setting up our library

Let's turn to our library entry point which we have seen needs to export the `Blog` struct that we call from our main function. As is standard we start with some import statements to make our subsequent code easier to work with:

src/lib.rs

```
#[macro_use]
extern crate diesel;
#[macro_use]
extern crate serde_derive;

use actix_web::{middleware, App, HttpServer};
use diesel::prelude::*;
use diesel::r2d2::{self, ConnectionManager};
```

Working with Diesel requires some macros to be in scope which is why we have the `macro_use` attribute on the `extern crate diesel` item. We also want to derive the `Serialize` and `Deserialize` traits from Serde for working with JSON so we bring those macros in to scope. The `actix_web` use statement is just bringing some items into scope to make our code more readable.

Diesel has a prelude which includes common types and functions which you almost always need while working with your database and the standard practice is to use the `*` import to bring all of the things exported in the prelude into scope as there is very little chance of a conflict.

As we discussed previously `r2d2` is a connection pooling library that Diesel provides an interface to because we turned that feature on in our manifest.

The next thing we do for code readability is to create a type alias for our pool of database connections:

src/lib.rs

```
15 type Pool = r2d2::Pool<ConnectionManager<SqliteConnection>>;
```

This is common practice as the exact details of this type are not usually relevant where we are using it, rather just the fact that it is a connection pool. Overusing type aliases can sometimes lead to more confusing code, but underusing them can also lead to noisy code. When to use one is a bit of art and style, but it makes a lot of sense here.

Modules and code organization

In previous chapters we have had all of our library code in `src/lib.rs` partially out of convenience and partially because the cost of spreading code across multiple files was higher than the benefit. Here we are going to be spreading our code out into multiple modules that together form our library. Each one has a clear purpose and the separation makes it easier to scale the codebase.

Rust has a module system to organize code within a crate for readability and ease of reuse as well as to control privacy of items. You can declare modules within a file with the `mod` keyword followed by a name, followed by curly braces to contain the code for the module:

```
mod a_module {
  fn some_function() {}
  fn hidden_function() {}

  mod inner_module {
    fn inner_function() {}
    fn inner_hidden_function() {}
  }
}

fn do_things() {
  a_module::some_function();
  a_module::inner_module::inner_function();
}
```

This code will not compile however. By default all items in Rust are private, so it is not possible to refer to any of the items inside `a_module` from outside because the lack of privacy specifier implies they are private. If we want the above code to compile, we would need to expose the functions to the outside. There are some technicalities to specifying privacy that allows you to make very fine grained judgements about how public an item actual is. We will show this through some examples as necessary, but otherwise it is not strictly necessary to know until you need it. The simplest fix to the above modules is to use the `pub` keyword to expose the items we want to expose:

```
mod a_module {
  pub fn some_function() {}
  fn hidden_function() {}

  pub mod inner_module {
    pub fn inner_function() {}
    fn inner_hidden_function() {}
  }
}
```

We can also spread modules over files and directories which helps with file size and makes working with large projects much easier. Instead of declaring a module inline, you can simply refer to it via:

```
mod a_module;
```

The language then expects to find a file `a_module.rs` or a file `mod.rs` inside a directory with the name of the module, i.e. `a_module/mod.rs`. Prior to the 2018 edition of Rust, if you wanted to have submodules of a module then you were required to use the directory plus `mod.rs` file method. In current Rust, you can have submodules within a directory and still have a file with the module name at the same level as the directory. In other words, our example above in modern Rust can have this form:

```
src/
├── a_module
│   └── inner_module.rs
├── a_module.rs
└── lib.rs
```

Or it can have this form (which was the only form prior to the 2018 edition):

```
src/
├── a_module
│   ├── inner_module.rs
│   └── mod.rs
└── lib.rs
```

There is a bit of flexibility even beyond this depending on the structure of the rest of the library around this, but this is what you will find most often in practice. We mention the `mod.rs` style as you will see it in other codebases that existed prior to the 2018 edition and did not migrate completely to the new style. It is not the recommended style any longer.

Returning back to our library, we specify the modules that make up our crate in `src/lib.rs`:

src/lib.rs

```
10 mod errors;
11 mod models;
12 mod routes;
13 mod schema;
```

We will have to create files and/or directories for these in order for this to compile, but we get these out of the way so they are part of the library. If you create a file in `src` but do not include it in the root module which is represented by `src/lib.rs` then that file will be completely ignored by Cargo and will not be part of your library. Therefore in order to actually make these modules part of our library we must include them in our root module.

Note also that these are all private. We currently are not exposing the existence of these modules to outside user's of our library. The decision to separate the code into modules is thus purely for the internal code structure and has not external impact.

The four modules we define are:

- errors
  - code for working with various failure scenarios
- models
  - code to define the Rust representation of our data model as represented by our database
- routes
  - code for defining the handlers that will make up the functions that get called by the framework in response to web requests
- schema
  - this is autogenerated by Diesel as we have mentioned before

The `src/schema.rs` file that we discussed previously now makes sense as a module that is part of our library which can be made available by including it here at the root. The macros used in that file are one of the reasons we imported them at the top of this file.

Application struct

We turn now to the `Blog` struct that we expose publicly and which we use from our main. This follows a very similar structure to the other applications we have developed. First we define a struct to hold some configuration data:

src/lib.rs

```
17 pub struct Blog {
18     port: u16,
19 }
```

For this again we only store the port that we are going to bind to. Then we create our implementation to add some functionality to our struct:

src/lib.rs

```
impl Blog {
    pub fn new(port: u16) -> Self {
        Blog { port }
    }

    pub fn run(&self, database_url: String) -> std::io::Result<()> {
        let manager = ConnectionManager::<SqliteConnection>::new(databa\
se_url);
        let pool = r2d2::Pool::builder()
            .build(manager)
            .expect("Failed to create pool.");

        println!("Starting http server: 127.0.0.1:{}", self.port);
        HttpServer::new(move || {
            App::new()
                .data(pool.clone())
                .wrap(middleware::Logger::default())
                .configure(routes::users::configure)
        })
        .bind(("127.0.0.1", self.port))?
        .run()
    }
}
```

In our `run` method we take a String representation of a database URL as input which we use to construct a pool of database connections. The `ConnectionManager` type is generic over the underlying connection which we specify as `SqliteConnection`. The `SqliteConnection` type is imported as part of the Diesel prelude because we turned the Sqlite feature on. We then use the `r2d2` factory method to construct a connection pool and store it in the `pool` variable.

Similar to our previous chapter where we created an app with state, we are going to follow the same pattern except here the type of the state is a connection pool. As the closure that gets passed to `HttpServer::new` is a factory we have to clone our `pool` so that each worker will have access to the same shared pool. It is an implementation

detail but the `Pool` type is just an `Arc` around a struct that manages connections so calling clone on the pool is the same as calling clone on an `Arc`, exactly how we managed state before.

The only other new piece here is the call to `configure` to set up the routes for users. We are passing a function to configure `routes::users::configure` which tells us that our `routes` module needs to publicly expose a submodule called `users`, and that submodule needs to publicly expose a function called `configure`. In the previous chapters we constructed all of our routes directly inside this factory function. Here we are using the configure method on `App` which takes one argument which satisfies the trait bound `FnOnce(&mut ServiceConfig)` which basically means a function that takes one argument of type `&mut ServiceConfig` and which we are only guaranteed that it is okay to call it once. When we get to the `users` module inside the `routes` module we will see how the `configure` method does the work of defining routes and handlers.

The choice to centralize this routing or to spread it out to separate modules is a matter of preference. There are trade-offs to both approaches, so although we demonstrate one style in this chapter, what you end up using will depend on your actual system.

We now take each of the modules that we previously declared in order to round out our library.

Errors

The first module we are going to deal with is the `errors` module. We define our own error type which unifies our notion of error states across different parts of the application. This encapsulates the different types of errors that can happen so we can explicitly handle those scenarios and can avoid generic 500 errors as much as possible. We also use this type to translate errors from other libraries to our own domain specific error type.

We create a new file at `src/errors.rs` which corresponds to the `mod errors;` statement in our root module. Let's first add some imports:

src/errors.rs

```
use actix_web::error::BlockingError;
use actix_web::web::HttpResponse;
use diesel::result::DatabaseErrorKind::UniqueViolation;
use diesel::result::Error::{DatabaseError, NotFound};
use std::fmt;
```

We are going to translate some `actix_web`errors and some `diesel` errors to our own custom type, so we bring in some types to reduce the amount of noise when refering to those errors. We are also going to describe how our error type can be converted into an HTTP response so we pull in parts of `actix_web` that are convenient for that purpose. Finally, we bring in the `std::fmt` module because we are going to implement a trait from within that module for our error type.

There are multiple ways of representing errors in Rust including a number of crates that make the definition of an error type easier. However, we are going to take the straightforward route of creating an `enum` for our error type. In this context encountering an error means we enter one of a small number of states where we want to return an error code and a message instead of continuing to process the request. This is a natural use case for an enumerated type. We define the type as:

src/errors.rs

```
#[derive(Debug)]
pub enum AppError {
    RecordAlreadyExists,
    RecordNotFound,
    DatabaseError(diesel::result::Error),
    OperationCanceled,
}
```

These are the only error states that we going to explicitly handle. As you build up your application and rely on more libraries that could fail you can add variants to this type to capture those errors if you want to deal with them this way. It is also possible to handle errors directly and not use this machinery which will allow us to return early with a custom status code and message.

The first two variants relate to errors returned from Diesel that we convert into easier to work with variants. We also have a catch-all `DatabaseError` for other Diesel errors that we don't specifically handle. We use a tuple struct to capture the underlying Diesel error for later use. Finally, `OperationCanceled` is related to a `actix_web` error having to do with an async operation which we will explain later.

First we will implement the `fmt::Display` trait for our type:

src/errors.rs

```
15 impl fmt::Display for AppError {
16     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
17         match self {
18             AppError::RecordAlreadyExists => write!(f, "This record vio\
19 lates a unique constraint"),
20             AppError::RecordNotFound => write!(f, "This record does not\
21  exist"),
22             AppError::DatabaseError(e) => write!(f, "Database error: {:\
23 ?}", e),
24             AppError::OperationCanceled => write!(f, "The running opera\
25 tion was canceled"),
26         }
27     }
28 }
```

We automatically implemented the `Debug` trait with the `derive` attribute on our struct which allows us to format instances of our type with the debug string formatter: `{:?}`. Implementing `Display` let's us print our type with `{}`. We must implement this trait because a different trait we want to implement requires `Debug` and `Display` to be implemented.

The implementation is pretty straightforward and most implementations of `Display` look like this. The macro `write!` is like `println!` except the first argument is a "Writer" and it returns a `Result` in the case of an error. The `println!` macro can panic in certain error scenarios rather than returning a `Result`. The `&mut fmt::Formatter` argument implements a trait that makes it a "Writer" so typically you just use `write!(f, ...)` and fill in the `...` with whatever you want to represent your type when it is formatted using `{}`.

From and Into

Rust usually does not implicit convert one type to another, but there is a mechanism for telling Rust how to convert between types which you can opt in to. There are two related traits: `From` and `Into`. You must explicitly implement one of these traits to be able to take advantage of some automatic type conversions. The other trait you do not implement can be derived as the two traits are, in a sense, inverses of each other. In the standard library and most code you will encounter only `From` is implemented and then `Into` is automatically satisfied for free.

One place that uses `From` is the `?` operator for early returning the `Err` variant of a `Result`. That is if the error that would be returned is type `X` and the expected return type is `Y` then you can still use the `?` operator if `Y` implements `From<X>`.

We are going to use this trait with our `AppError` type by implementing `From<X>` for a couple different values of `X` so that the `?` operator works without having to explicitly convert errors. The first conversion we are going to implement is for database errors:

src/errors.rs

```
26 impl From<diesel::result::Error> for AppError {
27     fn from(e: diesel::result::Error) -> Self {
28         match e {
29             DatabaseError(UniqueViolation, _) => AppError::RecordAlread\
30 yExists,
31             NotFound => AppError::RecordNotFound,
32             _ => AppError::DatabaseError(e),
33         }
34     }
35 }
```

So `From<diesel::result::Error> for AppError` means that you will be given an instance of `diesel::result::Error` and are expected to return an instance of `AppError`. We match on the Diesel error to handle the two specific cases we care about.

We see here the types that we imported from Diesel, in particular, when a unique constraint is violated at the database level, Diesel will convert this into a `DatabaseError(UniqueViola`

_) where the _ represents more data that we don't care about. We just care that whatever query we executed resulted in this specific type of error. We convert that to our `RecordAlreadyExists` variant as we will only get unique constraint violations when we try to insert a record that already exists based on what we have defined to be unique. Specifically, we have set a unique constraint on username so trying to insert two users with the same username will result in this `RecordAlreadyExists` error being created.

The second case is when we try to get a record from the database that does not exist. Diesel will return a `NotFound` error which we just turn into our variant with basically the same name.

Finally, the catch all case in the match statement means Diesel encountered an error other than these two and the only thing we know how to do is call it a `DatabaseError` and capture the underlying error if we maybe want to do something with it later. This is useful for debugging.

The encapsulation here means that we do not have to have any code destructing Diesel errors anywhere else in our code. We can just convert any Diesel error into our `AppError` and then only deal with that type in our code. This layer of abstraction allows us to reduce the surface area of necessary changes if we decide to change how we handle database errors across the entire application.

The next type we want to convert into our custom type is `BlockingError<AppError>`:

src/errors.rs

```
36 impl From<BlockingError<AppError>> for AppError {
37     fn from(e: BlockingError<AppError>) -> Self {
38         match e {
39             BlockingError::Error(inner) => inner,
40             BlockingError::Canceled => AppError::OperationCanceled,
41         }
42     }
43 }
```

`BlockingError<T>` is an actix web specific error that we will encounter when we get to the implementation of our handlers. We use this From‘ to do what is clearly a simple conversion lest we have to include code like this in every handler. Our

handlers will return futures but we must use blocking code to interact with the database. Therefore our handlers will run blocking code which can either succeed or can fail because the future was canceled or the underlying blocking code returned an error.

Errors as responses

The main advantage of creating our own error type is that we define how to turn an instance of `AppError` into an HTTP response and therefore automatically get nice error responses by just returning an error from a handler. As we are writing a JSON API, let's create a struct to represent a JSON error response:

src/errors.rs

```
45 #[derive(Debug, Serialize)]
46 struct ErrorResponse {
47     err: String,
48 }
```

JSON APIs should return JSON content or nothing as their error. Some APIs strangely return HTML data when errors occur because their framework does this by default. Don't write an API like that. We instead will return an object with one key `err` and a string value. You can make your own decision about how much or little information to return as part of an erroneous request.

Actix web defines a trait `ResponseError` which allows you to specify how the type inside a `Err` variant of a `Result` gets turned into a response. Let's implement it for `AppError`:

src/errors.rs

```
impl actix_web::ResponseError for AppError {
    fn error_response(&self) -> HttpResponse {
        let err = format!("{}", self);
        let mut builder = match self {
            AppError::RecordAlreadyExists => HttpResponse::BadRequest(),
            AppError::RecordNotFound => HttpResponse::NotFound(),
            _ => HttpResponse::InternalServerError(),
        };
        builder.json(ErrorResponse { err })
    }

    fn render_response(&self) -> HttpResponse {
        self.error_response()
    }
}
```

This trait is why we implemented `Display` for our error. First `ResponseError` has the trait bound `Debug + Display` which means that in order to implement `ResponseError` for your type, your type must also implement `Debug` and `Display`.

The trait requires `error_response` to be implemented which we do by matching on our error and setting useful response codes to the cases we care about and 500 otherwise, and then using the `Display` formatting to create an error message to return as JSON.

The trait also has a method `render_response` which has a default implementation, but the default overrides the content type and data which is not what we want. So we instead just implement this method to return the same thing as our `error_response` method which is what we want. When we get to our handler implementations we will see where this is called.

Models

The next module we are going to implement will be our layer that contains the interactions with the database. We will define the Rust representations of our data

model as well as functions for how to get those objects from the database. To get started we pull in some imports:

src/models.rs

```
use crate::errors::AppError;
use crate::schema::{users};
use diesel::prelude::*;
```

As we are going to be working with the database we pull in the Diesel prelude and we also bring in the `users` item from the autogenerated schema. We shall see how this is used to refer to different parts of the `users` database table.

In order to make our lives easier we are going to define our own `Result` type which will be an alias for `Result` in the standard library with the error type fixed as our `AppError` type:

src/models.rs

```
type Result<T> = std::result::Result<T, AppError>;
```

This way we can just return `Result<T>` and not have to writen `AppError` everywhere because throughout this module all errors will be of the `AppError` type so it is just noisy to have to write it everywhere.

User struct

We need a Rust struct to represent a user in the database. We create the struct just like any other:

src/models.rs

```
7  #[derive(Queryable, Identifiable, Serialize, Debug, PartialEq)]
8  pub struct User {
9      pub id: i32,
10     pub username: String,
11 }
```

Our users have `ids` which are `i32` because that maps to the database integer type and a `username` which is a `String` because the database column is a `VARCHAR`. We have seen the `Serialize`, `Debug`, and `PartialEq` trait derives before so we will not belabor those here, but `Queryable` and `Identifiable` are new.

`Queryable` is a trait that indicates this struct can be constructed from a database query. From the Diesel docs:

> Diesel represents the return type of a query as a tuple. The purpose of this trait is to convert from a tuple of Rust values that have been deserialized into your struct

So basically by deriving this trait we can automatically get a `User` struct from queries of the `users` table.

`Identifiable` is a trait that indicates that this struct represents a single row in a database table. It assumes a primary key named `id` but you can configure the derive attribute if you want to change the name of the primary key. It is required for associations which we will use later.

Create User

We have our model defined, let's get into actually talking to the database to create a user:

src/models.rs

```
pub fn create_user(conn: &SqliteConnection, username: &str) -> Result<U\
ser> {
    conn.transaction(|| {
        diesel::insert_into(users::table)
            .values((users::username.eq(username),))
            .execute(conn)?;

        users::table
            .order(users::id.desc())
            .select((users::id, users::username))
            .first(conn)
            .map_err(Into::into)
    })
}
```

Our function takes a SqliteConnection and a username string and returns either a User or an error. This function does not have to worry about where to get a database connection it simply assumes one and this let's us then interact with the database.

This code is slightly more complex because we are using Sqlite instead of a backend that supports a `RETURNING` clause. Sqlite does not support getting the `id` of a just inserted row as part of the insert statement. Instead we have to do another query to actually get the data back out to build a `User` struct. Because of this we run both queries inside a transaction to ensure that the logic of fetching the most recently inserted user actually returns the user that we just inserted. There are other approaches we could take to achieve the same end.

The connection type supports a method `transaction` which takes a closure. The closure must return a `Result`. If the `Result` is the error variant then the transaction is rolled back and the error result is returned. If instead we have the `Ok` variant then the transaction is committed and the successful result is returned.

Our first Diesel statement involves inserting a user with the supplied username into the `users` table. The nice thing about Diesel is that the code for doing this is readable and mostly self explanatory. The imported `users` item from the schema has an item for each column on the table as well as an item for the table as a whole, i.e.

`users::table`. There is a lot of machinery going on in the background to ensure that the types are correct based on what the database represents. Each database column has a unique type so you catch at compile time if you tried to insert the wrong type into a column rather than at runtime.

The `execute` method on the query builder returns a `Result` with the error being the Diesel specific error type. We can use the `?` operator here and return a `Result` with our `AppError` error type because of our `From` implementation in our errors module.

The second statement inside the transaction is fetching a single user from the database. We create a query where we order by descending `id` to get the most recently inserted row. As this is inside a transaction this select will execute logically after the insertion and therefore this will be the user we just inserted. We specify with the `select` method what columns we want in a tuple, in this case `id` and `username`. The `first` method tells Diesel to do a `LIMIT 1` and expect a single result. This therefore returns a `Result` with the successful variant being the resulting record or a not found error. The return type of our function is what gives Diesel enough information to convert the tuple that is returned into our `User` struct in addition to the fact that we derived `Queryable`.

Finally we call `map_err(Into::into)` which is a bit of a magical incantation. As we cannot use the `?` operator here to return our `Result` we have to explicitly convert the Diesel error type to our `AppError` to match the function signature. Rust does not automatically do this type conversion, however we can still use the type signature to help us. `map_err` is a method on `Result` which transforms the error variant with the supplied function and returns the success variant as is. We can pass the function `Into::into` to `map_err` and this uses the function signature to determine what to transform the error into. Furthermore, we can use `Into` here because we implemented `From` and `Into` gets implemented automatically.

Fetching a user

We can create users but this is not so fun unless we can also fetch a user from the database. We have two ways to identify a user: by `id` and by username. Rather than write two different functions for these use cases we are going to write one function which takes an enum that encapsulates which key to use for looking up the user. To this end, let's define an enum representing this key:

src/models.rs

```
27 pub enum UserKey<'a> {
28     Username(&'a str),
29     ID(i32),
30 }
```

Our enum is either an `ID` which holds an `i32` for looking up by the primary key or a `Username` which holds a reference to a string. We have seen the `'static` lifetime before but this is our first instance of a generic lifetime for a type we are creating.

As a referesher, lifetimes of references in Rust are checked by the compiler to ensure that the data referenced outlives the reference to it. In other words, the concept of lifetimes guarantees that we will not try to access memory after it has been deallocated while still being able to have access to data that we do not own. Lifetimes live in the same space as generic types and can be thought of as a kind of type. Here our type `UserKey<'a>` specifies that it has one generic lifetime parameter named `'a`. We need to specify this generic parameter so that we can give a definite lifetime to the string reference inside our `Username` variant.

Any composite type with a reference inside must declare a lifetime on that reference otherwise there is no way for the compiler to make any guarantees about the liveness of the underlying data. It is possible to use the special lifetime `'static` and not make our enum generic but that would force us to only be able to use static strings.

A different approach would be to declare the `Username` variant as `Username(String)`. This would most likely be just fine in this instance even if it resulted in an extra heap allocation (which depending on where we get the username from may or may not be required). However, making that decision at each juncture can lead to death by a thousand allocations. Instead, Rust gives you the tools to avoid the extra allocation while maintaining memory safety by ensuring that our reference is alive for at least as long as necessary.

Newcomers to Rust often use `String` and `clone` in many places that are unnecessary with just a bit of extra work after having been scarred by the borrow checker early on. Do not have a fear of heap allocations, but if you take a little extra time you might be surprised how many you can avoid by working with the compiler.

Now that we have a notion of a key to lookup a user by, we can write our function to actually get the user:

src/models.rs

```
32 pub fn find_user<'a>(conn: &SqliteConnection, key: UserKey<'a>) -> Resu\
33 lt<User> {
34     match key {
35         UserKey::Username(name) => users::table
36             .filter(users::username.eq(name))
37             .select((users::id, users::username))
38             .first::<User>(conn)
39             .map_err(AppError::from),
40         UserKey::ID(id) => users::table
41             .find(id)
42             .select((users::id, users::username))
43             .first::<User>(conn)
44             .map_err(Into::into),
45     }
46 }
```

We match on our key to decide which query to run. If we have a username then we do a filter by username equal to the value passed in. If we have an `id`, then we can use the special `find` function which attempts to find a single record based on the primary key for that table. Both queries use `first` to execute the query which as we said before either returns one instance of the type we are trying to return or returns a not found error.

Routes

Finally we are ready to get to our functions for handling requests by building out our third module, routes. This module is different than the rest in that it will have submodules. We create a `src/routes.rs` file which will be the entry point for the module, and then submodules will live inside a directory named `routes`.

Let's get some preliminaries out of the way for the base `routes` module before moving on to the `users` submodule. First, some imports:

src/routes.rs

```
use crate::errors::AppError;
use actix_web::HttpResponse;
```

Then we declare the users submodule:

src/routes.rs

```
pub(super) mod users;
```

This is similar to what we saw in the root module where we declared our other modules, e.g. `mod routes;`, however we also have the funny looking `pub(super)`.

You can declare an item as public with `pub` then it can be accessed externally from any module that can access all of the item's parent modules. If you want to restrict the visibility of an item to a specific scope then you can use one of `pub(in path)`, `pub(crate)`, `pub(super)`, or `pub(self)`, where `path` is a given module path. The visibility you choose depends on if and how you want an item to be exposed above the current module.

The module above `routes` is our root module at `src/lib.rs`. We want that module to be able to refer to the `users` module. However we do not want say our `models` module to be able to refer to the `users` module. So we restrict the visibility of the `users` module to only the module one step up in the hierarchy.

There is quite a bit of power in the Rust module system and for a long time it caused some confusion. The current system has been reworked to address some of that confusion. When you write a library it is important to decide what you really want to expose and be able to hide the rest as implementation details. Rust gives quite a bit of flexibilty in this regard and it is one of the features that makes growing Rust codebases more manageable.

> `pub(self)` is equivalent to nothing at all, i.e. `pub(self) mod foo` is the same as `mod foo` which is actually private. Why? The answer lies in macros that can generate code with visibility specifiers. If a macro outputs code like `pub($arg)` where `$arg`

> is an input argument, you might want to specify that the item should be private, so passing `self` as the argument achieves that goal.

The last part of our base routes module is a generic function which we will use to eliminate some repetitive code in our handlers:

src/routes.rs

```
fn convert<T, E>(res: Result<T, E>) -> Result<HttpResponse, AppError>
where
    T: serde::Serialize,
    AppError: From<E>,
{
    res.map(|d| HttpResponse::Ok().json(d))
        .map_err(Into::into)
}
```

This function takes some generic result and returns another result with fixed types which will end up being nice to return from our handler functions. We turn the success variant into a successful HTTP response with the data serialized as JSON. The error variant is turned into our `AppError` type which due to the work we did earlier can be returned from a handler and will result in a proper response with the relevant status code and JSON error message.

Now we can't just serialize any type into JSON nor can we convert any type to our `AppError`. We put trait bounds on the generic parameters to specify that we can only accept input arguments if the success variant is a type that can be serialized to JSON, i.e. `T: serde::Serialize`, and we can get an `AppError` from the error variant, i.e. `AppError: From<E>`. The syntax using the `where` after the function signature is sometimes necessary for complex trait bounds and is always possible to use. The other format of trait bound is directly in the specification of the names of the generic parameters between the name of the function and the opening parenthesis.

Given those trait bounds the implementation is pretty simple. We take the result and call map which operates only on the success variant and builds a response. The

`json` method on the response builder just requires that the argument passed can be serialized with Serde which is true given our bound. We chain with this call the invocation of `map_err` which operates only on the error variant. Here we rely on our `From` implementation and that definition of `Into::into` and this works because of the trait bound and the specified return type.

Routes for a user

Let's create our user module at `src/routes/users.rs` to fill in the module specified in our base routes module. First, as usual, some imports:

src/routes/users.rs

```
1  use crate::errors::AppError;
2  use crate::routes::convert;
3  use crate::{models, Pool};
4  use actix_web::{web, HttpResponse};
5  use futures::Future;
```

Note that our `convert` function in the routes module was not public but we are using it here. Private items are visible to the module they are defined in as well as all descendants.

Create a user

Our first handler is going to take a username as input and attempt to create a user. As we have seen before, we create a struct to represent the input data:

src/routes/users.rs

```
13 #[derive(Debug, Serialize, Deserialize)]
14 struct UserInput {
15     username: String,
16 }
```

As we derive `Deserialize` we will be able to accept this type as a JSON post body. We are now in place to write our handler:

src/routes/users.rs

```
fn create_user(
    item: web::Json<UserInput>,
    pool: web::Data<Pool>,
) -> impl Future<Item = HttpResponse, Error = AppError> {
    web::block(move || {
        let conn = &pool.get().unwrap();
        let username = item.into_inner().username;
        models::create_user(conn, username.as_str())
    })
    .then(convert)
}
```

We need the input data as the JSON body of the request and we need a handle to our database pool which we put inside our application state back in our application factory. The return type is new so let's unpack it.

`Future<Item, Error>` is a future in the traditional sense, an object that represents a computation which can be queried for a result or an error. This is a standard approach to writing asynchronous code where you return immediately some value that represents a computation rather than doing the computation before returning. Eventually the future resolves to a result or an error when the computation completes. Actix web is designed to work with both synchronous and asynchronous handlers, but so far we have only used the synchronous ones.

The syntax `impl Future` means that we are going to return some type that implements the `Future` trait, but we are not telling you exactly what that type is. This gives us some flexibilty and is necessary for some types which are hard (or impossible) to write.

As `Future` is a generic trait, we must fix the `Item` and `Error` types that are otherwise generic so that the return type is fully specified. The `Item` is the type that the future resolves to in a successful case. The `Error` is self explanatory. We want to return an `HttpResponse` in the successful case and our `AppError` in the other case.

Given all of the pieces we have built to get here, the handler itself is pretty straightforward. We get a connection to the database out of the pool. We get the

username from the input data. Given those two values we can use our `create_user` function in the `models` module to do the work. But what is `web::block`?

Diesel is synchronous, it does not directly support futures for interacting with the database. Therefore we use `web::block` which executes a blocking function on a thread pool and returns a future that resolves to the result of the function execution.

Finally, we can use our `convert` function to turn the result of the call to `models::create_user` into the response we desire. Note that here we see why we implemented `From<BlockingError<AppError>>` for our `AppError` type. The `map_err` function inside `convert` relies on that `From` implementation.

Find a user

We have two ways to find a user, by username or by id. Let's create a handler called `find_user` which implements the lookup by username:

src/routes/users.rs

```
30 fn find_user(
31     name: web::Path<String>,
32     pool: web::Data<Pool>,
33 ) -> impl Future<Item = HttpResponse, Error = AppError> {
34     web::block(move || {
35         let conn = &pool.get().unwrap();
36         let name = name.into_inner();
37         let key = models::UserKey::Username(name.as_str());
38         models::find_user(conn, key)
39     })
40     .then(convert)
41 }
```

The structure is pretty similar to our previous handler. This will be a GET request so we expect the username to be a string in the path. We have to create a `UserKey::Username` and then call our `models::find_user` function to do the work. The rest of the handler structure is the same as before.

Let's create `get_user` which does the lookup by id:

src/routes/users.rs

```
fn get_user(
    user_id: web::Path<i32>,
    pool: web::Data<Pool>,
) -> impl Future<Item = HttpResponse, Error = AppError> {
    web::block(move || {
        let conn = &pool.get().unwrap();
        let id = user_id.into_inner();
        let key = models::UserKey::ID(id);
        models::find_user(conn, key)
    })
    .then(convert)
}
```

This is basically the same thing except we expect an `i32` in the path instead of a string and we create the other variant of the `UserKey` enum.

Wiring the routes up

Recall the configuration of our app inside the application factory in `src/lib.rs`:

src/lib.rs

```
            App::new()
                .data(pool.clone())
                .wrap(middleware::Logger::default())
                .configure(routes::users::configure)
```

We call `configure` on the application builder with `routes::users::configure`. Let's define that function to get our user routes all hooked up:

src/routes/users.rs

```
pub fn configure(cfg: &mut web::ServiceConfig) {
    cfg.service(web::resource("/users").route(web::post().to_async(crea\
te_user)))
        .service(web::resource("/users/find/{name}").route(web::get().t\
o_async(find_user)))
        .service(web::resource("/users/{id}").route(web::get().to_async\
(get_user)));
}
```

The signature of this function is specified by Actix web. The only parameter is a mutable reference to a service configuration object. We saw in our earlier apps that we can create services that map URLs to routes which have methods and handlers. The object passed to us here allows for that exact same style of configuration.

We define three routes:

- POST `/users` which calls `create_user`
- GET `/users/find/{name}` which calls `find_user`
- GET `/users/{id}` which calls `get_user`

We use `to_async` to specify the handlers here because our handlers return futures rather than `to` that we used before with synchronous handlers.

Examples

We can test things out by running `cargo run` in one terminal and then using `curl` to try the endpoints that we just created.

Create a new user

The most basic functionality is posting JSON data to create a new user:

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/users -d '{"username":"Frank"}'

{
  "id": 1,
  "username": "Frank"
}
```

Create another new user

We can create yet another user and see that the auto incrementing ids in our database do what we expect:

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/users -d '{"username":"Bob"}'

{
  "id": 2,
  "username": "Bob"
}
```

Create a new user already exists

Let's see all of our error handling working by trying to create a user with a duplicate username:

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/users -d '{"username":"Bob"}'

{
  "err": "This record violates a unique constraint"
}
```

Lookup user by name

The next handler we created was to lookup a user by their username:

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/users\
/find/Frank

{
  "id": 1,
  "username": "Frank"
}
```

Lookup user by primary key

We should also be able to get a user by primary key:

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/users\
/1

{
  "id": 1,
  "username": "Frank"
}
```

Lookup user by name that doesn't exist

Again we can see our error handling working by trying to find a user that we did not create:

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/users\
/find/Steve

{
  "err": "This record does not exist"
}
```

Note that you can add the `-v` flag to Curl to see that status code for this response is 404 as a consequence of our implementation of `ResponseError` for `AppError`.

Extending our application

We have built quite a lot of parts that allows us to post some JSON data to create a user record and fetch users by username or primary key. With those pieces in place we can now easily add posts and comments. Posts are going to be our next step.

Extending the data model

The first step in adding a new data model is to modify our schema to support the new model. Let's get Diesel to generate a migration for us:

```
diesel migration generate create_posts
```

We can then add the necessary SQL for creating a posts table to `up.sql`:

migrations/2019-04-30-030252_create_posts/up.sql

```
CREATE TABLE posts (
  id INTEGER PRIMARY KEY NOT NULL,
  user_id INTEGER NOT NULL REFERENCES users (id),
  title VARCHAR NOT NULL,
  body TEXT NOT NULL,
  published BOOLEAN NOT NULL DEFAULT 0
)
```

Posts have a title and a body and a boolean specifying whether they are published or not. They also have a reference to a user which represents the author of the post. This syntax is particular to Sqlite so if you are using a different database this will have to change.

We also need to include the reverse of this migration in the corresponding `down.sql`:

migrations/2019-04-30-030252_create_posts/down.sql

```
DROP TABLE posts
```

Let's have Diesel run the migrations with `diesel migration run`. This command will modify the schema of the database as well as update our schema module to:

src/schema.rs

```
table! {
    posts (id) {
        id -> Integer,
        user_id -> Integer,
        title -> Text,
        body -> Text,
        published -> Bool,
    }
}

table! {
    users (id) {
        id -> Integer,
        username -> Text,
    }
}

joinable!(posts -> users (user_id));

allow_tables_to_appear_in_same_query!(
    posts,
    users,
);
```

The users table stays the same and the posts table appears as we would expect, however there are some new bits. First we see the `joinable!` macro which says sets up the necessary Rust types to allow posts and users to be joined. This is inferred

from the database schema because of the foreign key specification in our migration, i.e. `REFERENCES users (id)`.

There are debates about foreign key constraints in the database for certain data models and access patterns, but most of the time you want to let the database do what it is good at and enforce those invariants. Moreover, you get the automatic relationship mapping in Diesel if you specify the relationship at the database level.

The other new piece is the macro `allow_tables_to_appear_in_same_query!`. This is because we have more than one table now. It generates code that allows types from different tables to be mixed in the same query which is necessary for doing joins and sub-selects. It is not really necessary to understand the details of these macros to work with Diesel as it does the right thing most of the time. In exceptional circumstances you can consult the Diesel docs to find out more about the schema definitions.

Post model

We are going to expand on our models module. An alternative approach could be to split the models module into submodules for each model type similar to how we split routes into submodules. Where to draw these organizational lines is a matter of preference. For the models we decide on the single module approach because it makes working with the relationships between models a little bit simpler.

Our first step is to import posts from the generated schema:

src/models.rs

```
3 use crate::schema::posts;
```

Then we can introduce a struct that represents one post row in the database:

src/models.rs

```
#[derive(Queryable, Associations, Identifiable, Serialize, Debug)]
#[belongs_to(User)]
pub struct Post {
    pub id: i32,
    pub user_id: i32,
    pub title: String,
    pub body: String,
    pub published: bool,
}
```

Our database schema is directly translated to Rust types where each field is a column of the appropriate type. Note that the not null constraints we specified in our migration on each column is why we use types directly rather than wrapping them in `Option`. If we allowed nulls for some column then the type would have to be optional in Rust. We have also seen all of the traits that we are deriving before except for `Associations`, and the `belongs_to` attribute is new as well.

The concept of an association in Diesel is always from child to parent, i.e. there is no "has many" like in other ORMs. Declaring the association between two records requires the `belongs_to` attribute on the child and specifies the name of the struct that represents the parent. The `belongs_to` attribute accepts a `foreign_key` argument if the relevant foreign key is different from `table_name_id`. Both the parent and child must implement the `Identifiable` trait which we satisfy by deriving it. Finally, in addition to the `belongs_to` annotation, the struct also needs to derive `Associations`. Deriving this trait uses the information in `belongs_to` to generate the relevant code to make joins possible.

Creating a post

Let's now write the functions to work with posts in the database starting with creating one:

src/models.rs

```
pub fn create_post(conn: &SqliteConnection, user: &User, title: &str, b\
ody: &str) -> Result<Post> {
    conn.transaction(|| {
        diesel::insert_into(posts::table)
            .values((
                posts::user_id.eq(user.id),
                posts::title.eq(title),
                posts::body.eq(body),
            ))
            .execute(conn)?;

        posts::table
            .order(posts::id.desc())
            .select(posts::all_columns)
            .first(conn)
            .map_err(Into::into)
    })
}
```

We need a connection to the database and all of the relevant fields. We require a `User` object here which is a matter of choice. Instead we could have accepted a `user_id` directly. By default we set the post to be unpublished based on our schema, so we don't take a published parameter. The code for inserting into the posts table and then fetching the post is very similar to that for a user. Again the use of a transaction is a limitation of Sqlite not supporting a way to return data from an insert statement.

One difference here is the use of `select(posts::all_columns)` which is a shorthand that Diesel provides so that we do not have to write out a tuple with each column explicitly listed. Depending on the struct you are serializing to you may or may not be able to use this. If you always want a subset of columns it can be useful to write your own helper to represent a tuple of that subset.

Publish a post

As we stated our create method uses the database default for the published column and therefore in order to set published to true we need to update the database row

for a particular post. Let's create a function that takes the id of a post and sets the value of published to true:

src/models.rs

```
pub fn publish_post(conn: &SqliteConnection, post_id: i32) -> Result<Po\
st> {
    conn.transaction(|| {
        diesel::update(posts::table.filter(posts::id.eq(post_id)))
            .set(posts::published.eq(true))
            .execute(conn)?;

        posts::table
            .find(post_id)
            .select(posts::all_columns)
            .first(conn)
            .map_err(Into::into)
    })
}
```

This is our first instance of updating an existing row in the database. Issuing an update to the database uses the aptly named `update` function from Diesel. The argument to update can be a table, a filtered table (which is what we use here), or a reference to a struct that implements the `Identifiable` trait. If you pass just a table then the update applies to all rows of that table which is typically not what you want.

In this case we wish to only update the row with `id` equal to the `post_id` passed in as an argument. We are only updating a single column so we pass one expression to the `set` method, but if you want to update multiple columns you can pass a tuple to `set` instead.

> Diesel also has a trait called `AsChangeset` which you can derive which allows you to take a value like `post` and call `diesel::update(...).set(&post)` to set all of the fields (except the primary key) on the struct based on the current state of that

struct. See the Diesel update guide[a]) for more information.

[a] https://diesel.rs/guides/all-about-updates/

Retrieve posts

We are going to implement two different ways of retrieving posts. One to get all published posts and the other to get only those posts written by a particular user. First, let's get all posts:

src/models.rs

```
124 pub fn all_posts(conn: &SqliteConnection) -> Result<Vec<(Post, User)>> {
125     posts::table
126         .order(posts::id.desc())
127         .filter(posts::published.eq(true))
128         .inner_join(users::table)
129         .select((posts::all_columns, (users::id, users::username)))
130         .load::<(Post, User)>(conn)
131         .map_err(Into::into)
132 }
```

The return type of this function is a list of tuples where the first element is a post and the second element is the author. Diesel is built around queries that have this flat result structure. You might be used to other ORMs where a post object would have an author field which contains an embedded user object. In most uses of Diesel you will find tuples being used to represent related models rather than hierarchical structs.

We order the posts based on their id as this will make them newest first. Clearly a more robust model would include timestamps that we could sort by, but the idea is similar. We also select only those posts which have been published.

For each post we want to fetch all of the data about the post as well as data about the author. Hence we are finally getting to see a query involving multiple tables. To accomplish this we need to join with the users table. We can use `inner_join` with

just the name of the table and Diesel will figure out how to perform the join based on the association attributes we put on the `Post` model. The argument to `select` is a tuple with two elements both of which are tuples representing the columns we want to fetch. We then tell `load` the type to coerce these columns into and Diesel takes care of the rest.

The second way we are going to retrieve posts is only those authored by a particular user:

src/models.rs

```
134 pub fn user_posts(
135     conn: &SqliteConnection,
136     user_id: i32,
137 ) -> Result<Vec<Post>> {
138     posts::table
139         .filter(posts::user_id.eq(user_id))
140         .order(posts::id.desc())
141         .select(posts::all_columns)
142         .load::<Post>(conn)
143         .map_err(Into::into)
144 }
```

We have a choice here whether to take a `User` struct or just a `user_id` as input. If we chose the `User` struct route, this would require doing a database fetch somewhere else to reify the struct just to pull the id off inside this function. However that could be a design you want to adopt depending on how you want to handle the case of a `user_id` that does not exist. We choose the straight `user_id` method here which will just result in an empty set of posts if the user does not exist.

As the author is the same for all of these posts we only return a vector of posts rather than the tuple of our previous function. Other than that this function is much like our other fetching functions.

Adding routes for posts

With the data model in place and the necessary functions written for interacting with the database, we now can expose this functionality as part of our API by building the

necessary routes. First, we declare and export the soon to be written `posts` module within our `routes` module:

src/routes.rs

```
5 pub(super) mod posts;
```

Then we create the `routes/posts.rs` file to contain the code for our post routes. We start off with the same imports as our `routes::users` module:

src/routes/posts.rs

```
1 use crate::errors::AppError;
2 use crate::routes::convert;
3 use crate::{models, Pool};
4 use actix_web::{web, HttpResponse};
5 use diesel::prelude::*;
6 use futures::Future;
```

Creating a post

Our first task will be to create a post. As we have done before we create a struct to represent the JSON input data:

src/routes/posts.rs

```
18 #[derive(Debug, Serialize, Deserialize)]
19 struct PostInput {
20     title: String,
21     body: String,
22 }
```

We only need to get the title and body of the Post as the rest of the information will be inferred from the URL or will take on a default value. Let's next write the handler function for adding a post:

src/routes/posts.rs

```
fn add_post(
    user_id: web::Path<i32>,
    post: web::Json<PostInput>,
    pool: web::Data<Pool>,
) -> impl Future<Item = HttpResponse, Error = AppError> {
    web::block(move || {
        let conn: &SqliteConnection = &pool.get().unwrap();
        let key = models::UserKey::ID(user_id.into_inner());
        models::find_user(conn, key).and_then(|user| {
            let post = post.into_inner();
            let title = post.title;
            let body = post.body;
            models::create_post(conn, &user, title.as_str(), body.as_st\
r())
        })
    })
    .then(convert)
}
```

We will structure the URL so that the relevant `user_id` for the author will be part of the path. We take that path as input as well as the post as JSON and the database pool. This is much like our previous handlers so we will focus only on the unique bits here. We wrote our `create_post` function to take a user struct as input rather than just a plain id, therefore we need to convert the id we take as input into a `User` before we can use it. We do that so the error that results from a missing user happens first before we even try to create a post.

We use the `and_then` method on `Result` to continue on to creating a post only in the case where we actually found a user. This way we handle all of the different errors without having a mess of conditionals. We again build on all of the model code we wrote earlier so that the process of creating the post is mostly just plumbing the different parts together. In the end we again use our `convert` function to map the result into our expected form.

Publishing a post

The process of publishing a post in our API is quite simple, we just need a `post_id` in the path and can then rely on the model code we wrote earlier:

src/routes/posts.rs

```
42 fn publish_post(
43     post_id: web::Path<i32>,
44     pool: web::Data<Pool>,
45 ) -> impl Future<Item = HttpResponse, Error = AppError> {
46     web::block(move || {
47         let conn: &SqliteConnection = &pool.get().unwrap();
48         models::publish_post(conn, post_id.into_inner())
49     })
50     .then(convert)
51 }
```

Fetching posts

We can fetch posts either given a `user_id` or just fetch them all. In the first case we expect to find the id in the path:

src/routes/posts.rs

```
53 fn user_posts(
54     user_id: web::Path<i32>,
55     pool: web::Data<Pool>,
56 ) -> impl Future<Item = HttpResponse, Error = AppError> {
57     web::block(move || {
58         let conn: &SqliteConnection = &pool.get().unwrap();
59         models::user_posts(conn, user_id.into_inner())
60     })
61     .then(convert)
62 }
```

In the second case, we do not need any extra input to find all posts:

src/routes/posts.rs

```
fn all_posts(pool: web::Data<Pool>) -> impl Future<Item = HttpResponse,\
 Error = AppError> {
    web::block(move || {
        let conn: &SqliteConnection = &pool.get().unwrap();
        models::all_posts(conn)
    })
    .then(convert)
}
```

Wiring up our routes

As we did for users, we are going to export a `configure` function which modifies an input configuration to add the relevant routes for posts:

src/routes/posts.rs

```
pub fn configure(cfg: &mut web::ServiceConfig) {
    cfg.service(
        web::resource("/users/{id}/posts")
            .route(web::post().to_async(add_post))
            .route(web::get().to_async(user_posts)),
    )
    .service(web::resource("/posts").route(web::get().to_async(all_post\
s)))
    .service(web::resource("/posts/{id}/publish").route(web::post().to_\
async(publish_post)));
}
```

We add the four routes for the handlers we just wrote. Note that the path `/users/{id}/posts` accepts both a `POST` and a `GET` request which route to our `add_post` and `user_posts` handlers, respectively. Otherwise this is analogous to our configuration of the users routes.

Finally, we add a call to `configure` inside our application factory to actually add these routes to our app:

src/lib.rs

```
38                 .configure(routes::posts::configure)
```

Extending further: comments

The last piece of our puzzle is going to be adding comments. A comment will reference a user which represents the author of the comment and a post which is what the comment is commenting on. Adding this model is quite similar to adding posts so although we are going to show all of the necessary code changes, we will only explicate the elements which are novel

Modifying the model

Let's generate our migration:

```
diesel migration generate create_comments
```

In our `up.sql` we create a table for comments, the difference here being that comments will have foreign keys to both users and posts:

migrations/2019-04-30-031136_create_comments/up.sql

```
1  CREATE TABLE comments (
2    id INTEGER PRIMARY KEY NOT NULL,
3    user_id INTEGER NOT NULL REFERENCES users (id),
4    post_id INTEGER NOT NULL REFERENCES posts (id),
5    body TEXT NOT NULL
6  )
```

The `down.sql` just drops the table as we have seen before:

migrations/2019-04-30-031136_create_comments/down.sql

```
DROP TABLE comments
```

Generated schema

After running these migrations, we get our final schema:

src/schema.rs

```
table! {
    comments (id) {
        id -> Integer,
        user_id -> Integer,
        post_id -> Integer,
        body -> Text,
    }
}

table! {
    posts (id) {
        id -> Integer,
        user_id -> Integer,
        title -> Text,
        body -> Text,
        published -> Bool,
    }
}

table! {
    users (id) {
        id -> Integer,
        username -> Text,
    }
}

```

```
joinable!(comments -> posts (post_id));
joinable!(comments -> users (user_id));
joinable!(posts -> users (user_id));

allow_tables_to_appear_in_same_query!(comments, posts, users,);
```

The changes here from the previous schema are what we should expect by now. We have a table for comments and the relevant `joinable!` calls based on the foreign keys.

Comment model

Like for our other models, we add the import of the Diesel generated comments item from the schema:

src/models.rs

```
use crate::schema::comments;
```

Let's then create a struct to represent one row in the comments table:

src/models.rs

```
#[derive(Queryable, Identifiable, Associations, Serialize, Debug)]
#[belongs_to(User)]
#[belongs_to(Post)]
pub struct Comment {
    pub id: i32,
    pub user_id: i32,
    pub post_id: i32,
    pub body: String,
}
```

This is quite similar to our `Post` struct except we have an extra `belongs_to` attribute for the `User` association.

Creating a comment

We choose to take the `user_id` and `post_id` directly for convenience here as we create a function for creating a comment:

src/models.rs

```
pub fn create_comment(
    conn: &SqliteConnection,
    user_id: i32,
    post_id: i32,
    body: &str,
) -> Result<Comment> {
    conn.transaction(|| {
        diesel::insert_into(comments::table)
            .values((
                comments::user_id.eq(user_id),
                comments::post_id.eq(post_id),
                comments::body.eq(body),
            ))
            .execute(conn)?;

        comments::table
            .order(comments::id.desc())
            .select(comments::all_columns)
            .first(conn)
            .map_err(Into::into)
    })
}
```

The function here follows the same pattern as we have seen before for creating other models with again the transaction to support returning the newly created `Comment` struct due to the constraints of Sqlite.

Getting comments on a post

Let's write a function that takes a `post_id` and fetches all the comments on that post:

src/models.rs

```
pub fn post_comments(conn: &SqliteConnection, post_id: i32) -> Result<V\
ec<(Comment, User)>> {
    comments::table
        .filter(comments::post_id.eq(post_id))
        .inner_join(users::table)
        .select((comments::all_columns, (users::id, users::username)))
        .load::<(Comment, User)>(conn)
        .map_err(Into::into)
}
```

As the post is known we do not need to return the post with the comment, but we do want to return the user who made the comment. This looks quite similar to our function for fetching all posts where we join with the users table and select the relevant data.

> This is a point of API design with a REST-like system where you can decide how much or how little to include in your responses. This is a much larger topic than we can cover, but note that although we are making one choice, each application will need to consider their own trade-offs.

Getting all comments by a user

We are going to fetch all comments made by a particular user, but just fetching the comments alone would be lacking some important information, notably information about the post the comment is on. So we want to fetch the post for each comment, but we don't want to fetch all of the post data because that would be too much. Instead we are going to make a new struct to represent a subset of the post data that we want to fetch alongside each comment. Let's start with the struct for our post:

src/models.rs

```
#[derive(Queryable, Serialize, Debug)]
pub struct PostWithComment {
    pub id: i32,
    pub title: String,
    pub published: bool,
}
```

We have just enough data to identify the post and give some context but we don't fetch the entire body. We note that this derives `Queryable` but not `Identifiable`. This struct can be constructed from a query but it does not represent an entire row in a table.

Now let's right our function to get the comments for a particular user:

src/models.rs

```
pub fn user_comments(
    conn: &SqliteConnection,
    user_id: i32,
) -> Result<Vec<(Comment, PostWithComment)>> {
    comments::table
        .filter(comments::user_id.eq(user_id))
        .inner_join(posts::table)
        .select((
            comments::all_columns,
            (posts::id, posts::title, posts::published),
        ))
        .load::<(Comment, PostWithComment)>(conn)
        .map_err(Into::into)
}
```

We filter based on the passed in `user_id` and then join with the posts data to get the extra information about the posts. We use the select method to narrow down which columns from the posts table we need to construct our `PostWithComment` struct and then load a tuple which results in getting the return type we want which is a vector of

tuples where each element is a comment and some data about that post the comment is on.

This method is similar to all comments for a particular post except it is the other association. For that method as the user is a small object we just fetch entire user structs, but one would probably want to restrict that output as well as the user model grows.

Including comments in our post fetching functions

Now that we have comments as a model, we are going to go back and edit our two methods for fetching posts and incorporate comments into the results.

We start with the method for fetching all posts. The change in implementation of this function is driven by the change in the return type inside the `Result` from:

```
Vec<(Post, User)>
```

to:

```
Vec<((Post, User), Vec<(Comment, User)>)>
```

That is, instead of fetching a tuple of post and author, we want to get the post, author, and a vector of comments where we include the author of the comment alongside the comment itself. Looking at the type of each element of the vector by itself can help clarify what is going on:

```
((Post, User), Comments)
```

where

```
type Comments = Vec<(Comment, User)>;
```

Once you have a good sense for what we are trying to obtain, the code for accomplishing this will make more sense. Let's jump into it:

src/models.rs

```
pub fn all_posts(conn: &SqliteConnection) -> Result<Vec<((Post, User), \
Vec<(Comment, User)>)>> {
    let query = posts::table
        .order(posts::id.desc())
        .filter(posts::published.eq(true))
        .inner_join(users::table)
        .select((posts::all_columns, (users::id, users::username)));
    let posts_with_user = query.load::<(Post, User)>(conn)?;
    let (posts, post_users): (Vec<_>, Vec<_>) = posts_with_user.into_it\
er().unzip();

    let comments = Comment::belonging_to(&posts)
        .inner_join(users::table)
        .select((comments::all_columns, (users::id, users::username)))
        .load::<(Comment, User)>(conn)?
        .grouped_by(&posts);

    Ok(posts.into_iter().zip(post_users).zip(comments).collect())
}
```

We first perform the same query as before which gets all posts and their corresponding authors and set this as `posts_with_user`. We then use the `unzip` method on `std::iter::Iterator` which turns an iterator of pairs into a pair of iterators. In this case we turn `Vec<(Post, User)>` into `(Vec<Post>, Vec<User>)`. We can then fetch all of the comments that belong to those posts by passing a reference to that vector to `belonging_to` which we get from deriving `Associations` on `Comment`.

To associate the comments into chunks indexed by the posts we use the `grouped_by` method provided by Diesel. Note this does not generate a `GROUP BY` statement in SQL rather it is just operating on the data structures in memory of already loaded data. In the end this transforms a `Vec<(Comment, User)>` into `Vec<Vec<(Comment, User)>>`.

Finally, we can use the `zip` method on iterator to take all of these vectors and combine them into the output format we were looking for. `posts.into_iter().zip(post_users)` just turns `(Vec<Post>, Vec<User>)` back into `Vec<(Post, User)>`. The final

`zip(comments)` takes `Vec<(Post, User)>` and `Vec<Vec<(Comment, User)>>` and puts them together into a single vector of our desired return type.

With this in place, modifying the posts for a particular user to include comments is a straightforward exercise in the same vein:

src/models.rs

```
142 pub fn user_posts(
143     conn: &SqliteConnection,
144     user_id: i32,
145 ) -> Result<Vec<(Post, Vec<(Comment, User)>)>> {
146     let posts = posts::table
147         .filter(posts::user_id.eq(user_id))
148         .order(posts::id.desc())
149         .select(posts::all_columns)
150         .load::<Post>(conn)?;
151 
152     let comments = Comment::belonging_to(&posts)
153         .inner_join(users::table)
154         .select((comments::all_columns, (users::id, users::username)))
155         .load::<(Comment, User)>(conn)?
156         .grouped_by(&posts);
157 
158     Ok(posts.into_iter().zip(comments).collect())
159 }
```

Adding routes for comments

We have our model in place so now we turn to the handlers for our routes. First we declare and export our comments route module:

src/routes.rs

```
4 pub(super) mod comments;
```

We bring in the same imports that we have in our other routes modules:

src/routes/comments.rs

```
1 use crate::errors::AppError;
2 use crate::routes::convert;
3 use crate::{models, Pool};
4 use actix_web::{web, HttpResponse};
5 use diesel::prelude::*;
6 use futures::Future;
```

Creating a comment

As in our other cases, we need a struct to represent the JSON input for a comment:

src/routes/comments.rs

```
17 #[derive(Debug, Serialize, Deserialize)]
18 struct CommentInput {
19     user_id: i32,
20     body: String,
21 }
```

With that in place, we can create a handler for adding a comment:

src/routes/comments.rs

```
fn add_comment(
    post_id: web::Path<i32>,
    comment: web::Json<CommentInput>,
    pool: web::Data<Pool>,
) -> impl Future<Item = HttpResponse, Error = AppError> {
    web::block(move || {
        let conn: &SqliteConnection = &pool.get().unwrap();
        let data = comment.into_inner();
        let user_id = data.user_id;
        let body = data.body;
        models::create_comment(conn, user_id, post_id.into_inner(), bod\
y.as_str())
    })
    .then(convert)
}
```

We take just the id of the post as input in the path and as our model function for creating a comment just takes the post id as input we can call our function directly.

Again this is a design decision and we went this route here to show the opposite approach as we did for creating a post. In that function we fetched the user before creating the post, here we just try to create the comment given whatever post id we are given. If the database has foreign key constraints then passing a bad post id will result in an error at the database level. If the database does not support those constraints or you do not specify them then this would be a source of bugs if you did not otherwise validate the input. The design is up to you.

Getting all comments on a post

We can build on our previous functions to create a simple handler for getting all comments for a particular post:

src/routes/comments.rs

```
fn post_comments(
    post_id: web::Path<i32>,
    pool: web::Data<Pool>,
) -> impl Future<Item = HttpResponse, Error = AppError> {
    web::block(move || {
        let conn: &SqliteConnection = &pool.get().unwrap();
        models::post_comments(conn, post_id.into_inner())
    })
    .then(convert)
}
```

Getting all comments by a user

Similarly, given a user we can fetch all of their comments:

src/routes/comments.rs

```
fn user_comments(
    user_id: web::Path<i32>,
    pool: web::Data<Pool>,
) -> impl Future<Item = HttpResponse, Error = AppError> {
    web::block(move || {
        let conn: &SqliteConnection = &pool.get().unwrap();
        models::user_comments(conn, user_id.into_inner())
    })
    .then(convert)
}
```

Wiring the routes up

Finally, as we did for users and posts, we expose a configure function to connect our handlers to routes:

src/routes/comments.rs

```
pub fn configure(cfg: &mut web::ServiceConfig) {
    cfg.service(web::resource("/users/{id}/comments").route(web::get().\
to_async(user_comments)))
        .service(
            web::resource("/posts/{id}/comments")
                .route(web::post().to_async(add_comment))
                .route(web::get().to_async(post_comments)),
        );
}
```

Moreover, we add the call to `configure` for comments to our application factory:

src/lib.rs

```
                .configure(routes::comments::configure)
```

Examples

Create a post

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/users/1/posts -d '{"title":"Frank says hello","body":"Hello friends"\
}'
```

```
{
  "id": 1,
  "user_id": 1,
  "title": "Frank says hello",
  "body": "Hello friends",
  "published": false
}
```

Create a post

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/users/2/posts -d '{"title":"Bob is here too","body":"Hello friends, \
also"}'
```

```
{
  "id": 2,
  "user_id": 2,
  "title": "Bob is here too",
  "body": "Hello friends, also",
  "published": false
}
```

Publish a post

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/posts/1/publish
```

```
{
  "id": 1,
  "user_id": 1,
  "title": "Frank says hello",
  "body": "Hello friends",
  "published": true
}
```

Comment on a post

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/posts/1/comments -d '{"user_id":2,"body":"Hi Frank, this is your fri\
end Bob"}'
```

```
{
  "id": 1,
  "user_id": 2,
  "post_id": 1,
  "body": "Hi Frank, this is your friend Bob"
}
```

List all posts

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/posts
```


```
[
  [
    [
      {
        "id": 1,
        "user_id": 1,
        "title": "Frank says hello",
        "body": "Hello friends",
        "published": true
      },
      {
        "id": 1,
        "username": "Frank"
      }
    ],
    [
      [
        {
          "id": 1,
          "user_id": 2,
          "post_id": 1,
          "body": "Hi Frank, this is your friend Bob"
        },
        {
          "id": 2,
          "username": "Bob"
        }
      ]
    ]
  ]
]
```

See posts

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/users\
/1/posts
```


```
[
  [
    {
      "id": 1,
      "user_id": 1,
      "title": "Frank says hello",
      "body": "Hello friends",
      "published": true
    },
    [
      [
        {
          "id": 1,
          "user_id": 2,
          "post_id": 1,
          "body": "Hi Frank, this is your friend Bob"
        },
        {
          "id": 2,
          "username": "Bob"
        }
      ]
    ]
  ]
]
```


Publish other post

```
curl -s -H 'Content-Type: application/json' -X POST http://localhost:89\
98/posts/2/publish
```


```
{
  "id": 2,
  "user_id": 2,
  "title": "Bob is here too",
  "body": "Hello friends, also",
  "published": true
}
```


List all posts again

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/posts
```


```
[
  [
    [
      {
        "id": 2,
        "user_id": 2,
        "title": "Bob is here too",
        "body": "Hello friends, also",
        "published": true
      },
      {
        "id": 2,
        "username": "Bob"
      }
    ],
    [

    ]
  ],
```

```
  [
    [
      {
        "id": 1,
        "user_id": 1,
        "title": "Frank says hello",
        "body": "Hello friends",
        "published": true
      },
      {
        "id": 1,
        "username": "Frank"
      }
    ],
    [
      [
        {
          "id": 1,
          "user_id": 2,
          "post_id": 1,
          "body": "Hi Frank, this is your friend Bob"
        },
        {
          "id": 2,
          "username": "Bob"
        }
      ]
    ]
  ]
]
```


See users comments

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/users\
/2/comments
[
  [
    {
      "id": 1,
      "user_id": 2,
      "post_id": 1,
      "body": "Hi Frank, this is your friend Bob"
    },
    {
      "id": 1,
      "title": "Frank says hello",
      "published": true
    }
  ]
]
```

See post comments

```
curl -s -H 'Content-Type: application/json' http://localhost:8998/posts\
/1/comments
[
  [
    {
      "id": 1,
      "user_id": 2,
      "post_id": 1,
      "body": "Hi Frank, this is your friend Bob"
    },
    {
      "id": 2,
      "username": "Bob"
    }
```

```
  ]
]
```

Wrapping up

We built a database-backed web server with interrelated models of varying degrees of complexity. Nearly every web server with a database is a slight extension of the ideas here. The number of models will grow and the queries will get more complex, but the basic ideas are the same.

Hopefully the last few chapters have given you the confidence to build your next web application with Rust. The expressive type system, increasingly vibrant ecosystem, and top-notch performance make Rust a strong contender against any other platform.

What is Web Assembly?

Intro to Web Assembly

Modern web browsers expose a set of APIs related to the user interface and user interactions (DOM, CSS, WebGL, etc.) as well as an execution environment for working with these APIs which executes JavaScript. WebAssembly, abbreviated Wasm, is a type of code which was created to be run inside this browser execution environment as an additional language alongside JavaScript. The purpose is therefore not to replace JavaScript but rather to augment it in situations which have different constraints.

Wasm is a language but as its name implies it is akin to a low level assembly language which is not meant to be written by hand, in contrast with JavaScript. Rather Wasm is intended to be a compilation target for higher level languages. As we will discuss, the strongly typed nature of Wasm along with the low-level memory model, imply that the currently most suitable languages for compiling to Wasm are C++ and Rust.

As part of the specification of Wasm, no web specific assumptions are made, thus although the original intent was to introduce a low level language to the web, the design allows Wasm to be used in a variety of other contexts as well.

The primary goals of Wasm are safety, speed, efficiency, and portability. Wasm is safe as code is validated and executed in a sandboxed environment that guarantees memory safety. JavaScript in the browser is also executed in a sandboxed environment which is part of where the idea comes from. In more traditional contexts, languages that are translated to machine code and run directly on host hardware are harder to secure in this fashion. As Wasm is designed to be translated to machine code it provides near native performance. There are still a few performance penalties to the environment that differentiate Wasm from truly native code, but the gap is continuing to narrow as the platform matures.

The representation of Wasm code is designed so that the process of transmitting, decoding, validating, and compiling is streamable, parallelizable, and efficient. In

other words, a binary format was created for Wasm based on all of the learnings from the growth of JavaScript on the web over the past several decades.

Type System

Wasm has four value types, abbreviated `valtype`:

- i32
- i64
- f32
- f64

These types represent 32 and 64 bit integers, as well as 32 and 64 bit floating point numbers. These floating point numbers are also known as single and double precision as defined by IEEE 754. The integer types are not signed or unsigned in the spec even. Do not be confused by the fact that the term `i32` is the Rust syntax for a signed 32 bit integer. As we will see later we can use either unsigned or signed integer types in Rust code, e.g. both `i32` and `u32`, with the signedness in Wasm inferred by usage.

Wasm has functions which map a vector of value types to a vector of value types:

```
function = vec(valtype) -> vec(valtype)
```

However, the return type vector is currently limited to be of length at most 1. In other words, Wasm functions can take 0 or more arguments and can either return nothing or return a single value. This restriction may be removed in the future.

Memory

Wasm has a linear memory model which is just a contiguous vector of raw bytes. Your code can grow this memory but not shrink it. Data in the memory region is accessed via load and store operations based on an aligned offset from the beginning of the memory region. Access via an offset from the beginning of the region is where the linear name comes from. This memory region is exposed by a Wasm module and can be accessed directly in JavaScript. Sharing memory is dangerous but is the primary way by which JavaScript and Wasm can interact performantly.

Execution

The WebAssembly computational model is based on a stack machine. This means that every operation can be modeled by maybe popping some values off a virtual stack, possibly doing something with these values, and then maybe pushing some values onto this stack. Each possible operation is well-defined in the Wasm specification as to exactly how a specific opcode interacts with the stack. For example, imagine we start with an empty stack and we execute the following operations:

```
1 i64.const 16
2 i64.const 2
3 i64.div_u
```

The first operation has two parts, `i64.const` is an opcode which takes one argument, `16` in this case. Thus, `i64.const 16` means push the value `16` as an `i64` constant onto the top of the stack. Similarly the second operation pushes `2` onto the stack leaving our stack to look like:

```
1 2
2 16
```

The next operation `i64.div_u` is defined to pop two values off the top of the stack, perform unsigned division between those two values and push the result back onto the top of the stack. In this case the first number popped off divides the second number popped off, so at the end of this operation the stack contains:

```
1 8
```

Rust in the browser

Rust uses the LLVM project as the backend for its compiler. This means that the Rust compiler does all of the Rust specific work necessary to build an intermediate representation (IR) of your code which is understood by LLVM. From that point,

LLVM is used to turn that IR into machine code for the particular target of your choosing.

Targets in LLVM can roughly be thought of as architectures, such as `x86_64` or `armv7`. Wasm is just another target. Hence, Rust can support Wasm if LLVM supports Wasm, which luckily it does. This is also one of the ways that C++ supports Wasm through the Clang compiler which is part of the LLVM project.

Rust installations are managed with the `rustup` tool which makes it easy to have different toolchains (nightly, beta, stable) as well as different targets installed simultaneously. The target triple for `wasm` is `wasm32-unknown-unknown`. The target triples have the form `<arch>-<vendor>-<sys>`, for example `x86_64-apple-darwin` is the default triple for a modern Macbook. The `unknown` vendor and system mean to use the defaults for the specific architecture. We can install this by running:

```
rustup target add wasm32-unknown-unknown
```

The Smallest Wasm Library

Let's create a Wasm library that does absolute nothing but generate a Wasm library. We start off by creating a library with Cargo:

```
cargo new --lib do-nothing
```

We need to specify that our library will be a C dynamic library as this is the type of compilation format that Wasm uses. Depending on the target architecture this is also how you would build a `.so` on Linux or `.dylib` on Mac OS. We do this by adding the `lib` section to our Cargo.toml:

Cargo.toml

```
[package]
name = "do-nothing"
version = "0.1.0"
authors = ["Your Name <you@eample.com>"]
edition = "2018"

[lib]
crate-type = ["cdylib"]
```

We can then remove all of the code in `src/lib.rs` to leave a blank file. This technically is a Wasm library, just with no code. Let's make a release build for the Wasm target:

```
cargo build --target wasm32-unknown-unknown --release
```

By default, Cargo puts the build artifacts in `target/<target-triple>/<mode>/` or `target/<mode>` for the default target, which in this case is `target/wasm32-unknown-unknown/rele` Inside that directory, we have a `do_nothing.wasm` file which is our "empty" Wasm module. But how big is it:

```
ls -lh target/wasm32-unknown-unknown/release/do_nothing.wasm
-rwxr-xr-x  2 led  staff   1.4M Jul  8 21:55 target/wasm32-unknown-unkn\
own/release/do_nothing.wasm
```

1.4M to do nothing! That's insane! But it turns out it is because the output binary still includes debug symbols.

Stripping out debug symbols can be done with a tool called `wasm-strip` which is part of the WebAssembly Binary Toolkit (WABT)[39]. The repository includes instructions on how to build that suite of tools. Assuming you have that installed somewhere on your path, we can run it to strip our binary:

[39] https://github.com/WebAssembly/wabt

```
wasm-strip target/wasm32-unknown-unknown/release/do_nothing.wasm
```

We can then check the size of our new binary:

```
ls -lh target/wasm32-unknown-unknown/release/do_nothing.wasm
-rwxr-xr-x  2 led  staff   102B Jul  8 22:01 target/wasm32-unknown-unkn\
own/release/do_nothing.wasm
```

A much more reasonable 102 bytes. We can make this better by running one more tool which is to actually optimize the binary, `wasm-opt` as part of the Binaryen library[40]. Again this repository has instructions for how to build and install this suite of tools. Running this against our binary requires us to specify a new file to put the output:

```
wasm-opt -o do_nothing_opt.wasm -Oz target/wasm32-unknown-unknown/relea\
se/do_nothing.wasm
```

Let's check the size of this optimized binary:

```
ls -lh do_nothing_opt.wasm
-rw-r--r--  1 led  staff    71B Jul  8 22:04 do_nothing_opt.wasm
```

We got down to 71 bytes. That is about as good as we can do. It is possible to manually shave a few more bytes off, but for all intents and purposes this is the baseline that we will build up from. This is incredibly small in the JavaScript ecosystem, but we are also not doing anything.

We are not going to spend too much time on optimizing the size of the Wasm binaries in subsequent things that we build, but it is important to use these tools for production use cases lest your binaries become unmanageable. We will point out certain features which dramatically increase binary size, but some of these are necessary for many applications.

Working with primatives

We are going to build upon our library that does nothing in small steps to understand how Wasm operates. The first step is to expose a function which only deals with primative values. Let's create a new library:

[40] https://github.com/WebAssembly/binaryen

```
cargo new --lib do-addition
```

Again we specify that we want our library to be a `cdylib` in Cargo.toml:

Cargo.toml

```
[package]
name = "do-addition"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[lib]
crate-type = ["cdylib"]
```

We expose a single function `add` which takes two unsigned integers and returns their sum as an unsigned integer:

src/lib.rs

```
#[no_mangle]
pub extern "C" fn add(a: u32, b: u32) -> u32 {
    a + b
}
```

The `#[no_mangle]` attribute tells the Rust compiler that we want the name of our function to be `add` in the final binary instead of some more complicated name that is auto-generated based on the name and types. Both Rust and C++ use name mangling for managing certain language features that are easier to implement if everything has a unique name. Usually you don't have to worry about the exact name of your functions in the compiled executable, but because we are exposing a library which will be callable from JavaScript we need to know the actual name we need to call. Without this attribute, we would end up with something like `N15do_addition_4a3b56d3add3` as the name of the add function.

We also put the modifier `extern "C"` on the function to say that we want this function to use the right calling conventions that Wasm will understand. Otherwise this is just a simple publicly exposed Rust function.

To make the process of building things easier we show one possible build script which builds our crate and performs all of the optimization steps we talked about in the previous section:

build.sh

```
#!/bin/bash

WABT_BIN=$HOME/Code/wabt/bin
BINARYEN_BIN=$HOME/Code/binaryen/bin
TARGET=wasm32-unknown-unknown
NAME=do_addition
BINARY=target/$TARGET/release/$NAME.wasm

cargo build --target $TARGET --release
$WABT_BIN/wasm-strip $BINARY
mkdir -p www
$BINARYEN_BIN/wasm-opt -o www/$NAME.wasm -Oz $BINARY
```

We then make this script executable:

```
chmod +x build.sh
```

We can then build and optimize our Wasm module with `./build.sh`. Let's check the size of our final binary in this first example where we are actually doing something:

$ ls -lh do_addition.wasm -rw-r–r– 1 led staff 101B Jul 16 22:45 do_addition.wasm

We are up some small number of bytes from our function which did nothing, as expected, but at 101 bytes this is still quite small.

Let's build up an HTML shell that will host our Wasm module and see our code working. The simplest thing is to have an HTML page which loads our Wasm code, uses it, and prints something to the JavaScript console:

www/index.html

```
<!DOCTYPE html>
<script type="module">
  async function init() {
    const { instance } = await WebAssembly.instantiateStreaming(
      fetch("./do_addition.wasm")
    );

    const answer = instance.exports.add(1, 2);
    console.log(answer);
  }

  init();
</script>
```

We must use an `async` function as all Wasm code must be asynchronously loaded at the time of this writing. Most likely this will remain true going forward. We write a function `init` which encapsulates the things we want to do and then call that as the last thing we do in our script.

The WebAssembly module in JavaScript exposes APIs for working with Wasm files and types, but the easiest to use is `instantiateStreaming` which let's us fetch our Wasm code and turn it into a module which we can interact with from JavaScript. This function returns a result object which contains an `instance` object and a `module` object. The module is useful if you want to share the compiled code with other contexts like Web Workers. For our purposes, we just care about the instance as it contains references to all the exported Wasm functions.

Our `add` function can be found as a property on the exports property of the instance object. Therefore, we can treat `instance.exports.add` like a JavaScript function which takes two integers and returns an integer, when really it is backed by native machine code generated from Rust.

Opening most browsers with this HTML page will not work because of browser restrictions on loading Wasm from local files. The easiest workaround is to serve your Wasm code and use that server to view your example. If you have Python installed on your system, you can run a simple server with:

serve.py

```
#!/usr/bin/env python3

import http.server
import socketserver

PORT = 8080

Handler = http.server.SimpleHTTPRequestHandler

Handler.extensions_map[".wasm"] = "application/wasm"

httpd = socketserver.TCPServer(("", PORT), Handler)

print("serving at port", PORT)
httpd.serve_forever()
```

Note the one change from the simplest Python file server is to explicitly set the MIME type for `.wasm` files. This makes sure browsers handle these files correctly.

We make this script executable as well:

```
chmod +x serve.py
```

And can then run it to serve our files:

```
./serve.py
```

Navigate to `http://localhost:8080/www` and look in the console to see Rust running in the browser:

```
3
```

That's really all there is to it. The Rust compiler knows how to turn your Rust code into the Wasm format. The JavaScript engine in the browser knows how to load

Wasm code. And, finally, there is some execution engine in the browser which an execute that Wasm code when called via the JavaScript interface. The one caveat is that the only Rust functions which are valid to expose to Wasm only deal with integers and floating point numbers.

Working with complex types

Being limited to integers and floating point numbers might seem like a pretty severe limitation, but any complex type can be built on top of these primitives with some effort and hopefully help from some tools. In order to understand the tools that make this easy we are first going to go the hard route and manually expose functions that operate on strings. This is not intended for production use, but is to make the tools we will show in the next section less magical.

Let's start by creating a new Rust library:

```
cargo new --lib hello-raw
```

Again we make our crate generate a dynamic library:

Cargo.toml

```
[package]
name = "hello-raw"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[lib]
crate-type = ["cdylib"]
```

Our goal is to expose the function `greet` which takes a `&str` as input and returns a new heap allocated `String`:

src/lib.rs

```
4  pub extern "C" fn greet(name: &str) -> String {
5      format!("Hello, {}!", name)
6  }
```

Note that we make it public and specify the `extern "C"` modifier to ensure it has the right calling convention. However we did not specify `#[no_mangle]` because we are not going to call this directly and therefore we want Rust to mangle the name so that the wrapper that we do expose can use the name `greet`.

Webassembly does not understand `&str` or `String` types, it can just take `0` or more numeric inputs and return a single numeric output. So we have to figure out how to turn `&str` into `1` or more numeric inputs and `String` into a single numeric output. There are actually many different ways to do this, each with trade-offs, and we are going to demonstrate one approach.

So let's instead write a greet function which takes two integers as input and returns one integer:

src/lib.rs

```
8  #[export_name = "greet"]
9  pub extern "C" fn __greet_wrapper(
10     arg0_ptr: *const u8,
11     arg0_len: usize,
12 ) -> *mut String {
13     let arg0 = unsafe {
14         let slice = std::slice::from_raw_parts(arg0_ptr, arg0_len);
15         std::str::from_utf8_unchecked(slice)
16     };
17     let _ret = greet(arg0);
18     Box::into_raw(Box::new(_ret))
19 }
```

There is a lot going on in this code so we will go through it in detail but first let's talk at a high level about what is going on. We have turned the string input into a pointer (which is just an integer) to the beginning of where the string lives in memory and

the length of the string (also an integer). Inside the function we do some work to turn that input into a `&str` which we then pass to our original `greet` function. We then do some work to take the `String` output of `greet` and turn it into a pointer which again is just an integer. So we have a function which only uses integer inputs and output but effectively exposes the higher level `greet` function we wrote that operates on strings.

Let's get into the details by first discussing the input and output types of our wrapper function. Pointers are not common in most of Rust because you usually use references when sharing data. However, when working across FFI (Foreign Function Interface) boundaries, e.g. when talking to JavaScript, we use pointers directly as they are a concept shared on both sides. We take a pointer to a memory region containing bytes (a `u8` value is just one byte) which represents a string and the length of that string. It is the responsibility of the caller to ensure that this pointer is valid and that the length is correct.

The return type is a pointer to a mutable String, i.e. a heap allocated String. Recall that String itself is already on the heap so this is an extra allocation. It is an implicit contract of our function that the caller is responsible for making sure that this pointer gets passed back to Rust at some point to clean up the memory on the heap. By returning a mutable pointer we are effectively saying you own this thing that we created because mutability in Rust comes from being the sole owner of an object. This is the same contract as our original `greet` function as the caller becomes the owner of the returned String. This concept is just a bit more implicit due to the FFI boundary.

The first step in our function is to take the pointer and length and turn it into a string slice, i.e. a `&str`. This is inherently an unsafe operation as the compiler has no mechanism to verify that the pointer actually points to useful data or that the length is correct. We use an `unsafe` block to tell the compiler we know it cannot check this for us but we are guaranteeing that the relevant invariants are satisfied. The standard library provides a function `std::slice::from_raw_parts` which will give us a `&[u8]`. This function is marked as unsafe which is why we have to call it inside an unsafe block.

The unsafe keyword does not turn off a lot of the checks the compiler does, but it does allow you to perform operations that it is impossible for it to verify. This definitely can lead to undefined behavior if you violate invariants you presumed to

be satisfying.

Given that we have a `&[u8]` we want to turn this into a `&str` which is really just a fancy name for a `&[u8]` which is also valid UTF-8. There are two mechanism for doing this `std::str::from_utf8` and `std::str::from_utf8_unchecked`. The former performs validity checks on the input to ensure that the passed in slice really is UTF-8. The latter just says trust me it is valid UTF-8 don't bother checking. The former returns `Result<&str, Utf8Error>` while the latter just returns `&str`. We are therefore adding one more implicit contract to our exposed function, the pointer we get must be to a sequence of bytes, the length must match the length of that sequence, and the sequence of bytes must be valid UTF-8. If any of those are violated then undefined behavior can result.

We could have used the safe conversion method, but it is unclear what we would do in the failure scenario. One possible answer is to throw a JavaScript exception. We have chosen to make the implicit assumption of valid UTF-8 input, but that choice is up to you.

Now that we converted the input parts into a string slice we no longer have to do anything unsafe and can continue in normal, safe Rust. This is the canonical way to use unsafe code in Rust. Keep the block as small as possible to get back into safe Rust quickly. This makes auditing the code and understanding the necessary invariants much more manageable. If you run into strange behavior it is possible to focus your debugging efforts on what is going on in the unsafe blocks. If these are small and contained then at least you have a chance.

We proceed to call our original `greet` function to get the relevant String back. We then need to somehow give a pointer to this String to the caller and ensure that the underlying memory is not deallocated. We use `Box::new` to create a new heap allocation to hold the String. Note that the String is already on the heap by its nature, but we create a new heap allocation to own the String because we can keep the String alive if we can keep the `Box` alive as it becomes the owner of the String. We do this by using the associated function `Box::into_raw`. This function consumes the `Box` and returns exactly the raw pointer we want.

The documentation for `Box::into_raw` is clear that whoever gets this raw pointer is responsible for managing the memory and must do something to ensure the box gets destroyed when it is no longer needed, otherwise you will leak memory. Note that this is not unsafe, leaking memory is not unsafe just undesirable. The Rust compiler

makes no guarantees about memory leaks and in fact you need to explicitly "leak" memory this way to accomplish some tasks. We will show later how we handle this and make sure that we clean up after ourselves. But it is important to note the underlying String is kept alive because the Box that owns the String is now owned by whoever called our function.

We will have more Rust to write, but in order to motivate it, let's turn now to the JavaScript side of things. We also need to write something of a wrapper function in JavaScript as it would not be a nice API to have to figure out how to turn your JavaScript string into a pointer and a length. Rather we want our consumers to call a function with a string and get back a string and not even need be aware that there is Rust running underneath. Let's write the `greet` function in JavaScript:

www/index.html

```
9     function greet(arg0) {
10      const [ptr0, len0] = passStringToWasm(arg0);
11      try {
12        const retptr = wasm.greet(ptr0, len0);
13        const mem = new Uint32Array(wasm.memory.buffer);
14        const rustptr = mem[retptr / 4];
15        const rustlen = mem[retptr / 4 + 1];
16        const realRet = getStringFromWasm(rustptr, rustlen).slice();
17        wasm.__boxed_str_free(retptr);
18        return realRet;
19      } finally {
20        wasm.__free(ptr0, len0);
21      }
22    }
```

This is much more complicated that our previous interaction with WebAssembly. The first step is to call a helper function that takes our input argument and turns it into the pointer and length that we can call our Wasm function with. We will define this shortly, but take it for granted for now but do know that this function puts the JavaScript string we took as an argument into the linear memory space shared between JavasScript and Wasm.

We setup a try/finally block to ensure that the string that was just moved into the Wasm memory always gets freed even if something funky happens while we are interacting with our exposed Wasm function. Inside the try block, we first call the greet function that we exposed with the pointer and length we got and declare a local variable to hold the pointer to the String we get back. This pointer points to a chunk of memory inside the Wasm linear memory region.

We can get a handle to the memory region shared with Wasm via the `buffer` property on the `memory` property that is exposed on our Wasm instance. The `memory` property is an instance of a `WebAssembly.Memory` object. This object can do two things, one is grow the memory region via a `grow` method, the other is give a handle to the underlying memory via the `buffer` property.

JavaScript has a variety of typed arrays for exposing an array-like view of an underlying buffer of binary data. We can construct an array which contains unsigned 32-bit integers (exactly what the 32-bit Wasm memory region is) by passing the Wasm buffer into the `Uint32Array` constructor. This is the fundamental concept for working with the shared memory region between JavaScript and Wasm. You create a typed array with the shared buffer and then read or write to this typed array. Wasm can also read and write this buffer so there be dragons here.

The pointer that is returned from the Wasm function is an offset from the beginning of the memory region, due to the return type and the fact that the array view over the memory buffer has type `Uint32` we need to do this little bit of arithmetic to get the array index for the pointer to the underlying bytes and the index of the length of the returned String. This works because in Rust a String is just a `Vec<u8>` and a `Vec<u8>` is a struct with two fields, the first being roughly a pointer to a buffer of `u8` values, and the second is a length. So we use that fact here to know that the pointer to the String is really a pointer to a buffer of bytes and a length.

Given a pointer to some bytes and a length, we use another utility function to extract a string from the Wasm memory region into JavaScript. We use `slice` to make a copy in JavaScript so we can safely tell Rust to free the memory it is currently using for that string. The following line where we call `__boxed_str_free` tells us that we need to implement this function over in Rust which will take a `*mut String` and do whatever is necessary to free it. This is how we hold up JavaScript's end of the bargain to not leak memory.

Finally, we can return the JavaScript string that results from this whole process.

No matter what happens in this whole process, if something goes wrong or not, the `finally` block will execute and free the memory in the Wasm address space associated with the argument. We used `passStringToWasm` to move the argument into the Wasm memory so we use `__free` to let it go. Again we see that we need to implement `__free` in Rust for this to work.

Before returning to Rust, let's implement the two helpers we need in JavaScript. First, let's take care of moving a JavaScript string into the Wasm memory region and returning a pointer into that region and a length:

www/index.html

```
24      function passStringToWasm(arg) {
25        const buf = new TextEncoder('utf-8').encode(arg);
26        const len = buf.length;
27        const ptr = wasm.__malloc(len);
28        let array = new Uint8Array(wasm.memory.buffer);
29        array.set(buf, ptr);
30        return [ptr, len];
31      }
```

JavaScript provides an API for turning a string into a memory buffer via a `TextEncoder` object. We say that we want the string encoded as UTF-8 to conform to the invariants in our Rust code. We then ask our Wasm instance to allocate memory of this particular size via a call to `__malloc`. We will need to write this. Finally, we create a `Uint8Array` with the Wasm memory buffer so that we can set the newly allocated memory to the bytes of our string. We then return the pointer and length which is all that is needed to reference the string in the Wasm memory region.

Lastly, let's write the function that will get us a JavaScript string out of the Wasm memory region:

www/index.html

```
33      function getStringFromWasm(ptr, len) {
34        const mem = new Uint8Array(wasm.memory.buffer);
35        const slice = mem.slice(ptr, ptr + len);
36        const ret = new TextDecoder('utf-8').decode(slice);
37        return ret;
38      }
```

This is mostly the reverse process of putting the string into the memory region. We get a `Uint8Array` view of the Wasm memory and then use `slice` to copy the bytes into a new typed array specified by the starting and ending points. This `slice` method on typed arrays is semantically the same as the one on normal JavaScript arrays. Given this array of bytes we use a `TextDecoder` to turn it into a string, again assuming that the bytes represent a UTF-8 string.

Okay so if we expose `__malloc`, `__free`, and `__boxed_str_free` in our Rust library with the correct semantics then our Wasm module and our JavaScript code will work in concert to properly call a Rust function from JavaScript that takes a string and returns a string.

To support the implementation of memory allocation and deallocation functions, we pull in some imports:

src/lib.rs

```
1  use std::alloc::{alloc, dealloc, Layout};
2  use std::mem;
```

Let's start with the hardest function and get progressively easier. First up then is `__malloc`:

src/lib.rs

```
#[no_mangle]
pub extern "C" fn __malloc(size: usize) -> *mut u8 {
    let align = mem::align_of::<usize>();
    if let Ok(layout) = Layout::from_size_align(size, align) {
        unsafe {
            if layout.size() > 0 {
                let ptr = alloc(layout);
                if !ptr.is_null() {
                    return ptr
                }
            } else {
                return align as *mut u8
            }
        }
    }

    panic!("malloc failed")
}
```

First the signature should be expected by now, we want this publicly accessible, we want it to use the "C" calling convention, and we want the name to be `__malloc` so we specify `#[no_mangle]`. We take a size as input (we use the byte length of our string) and return a pointer to some allocated bytes of that size.

Although you rarely have to deal with these low level APIs, as Rust is a systems language, it provides the facilities for working with memory layouts and proper alignment. The first thing we do is get the minimum alignment for a `usize` based on the ABI. We need this to pass to the Layout constructor because in order to allocate memory you need both a size and an alignment. Properly aligning our memory is necessary for a variety of reasons but suffice it to say that Rust takes care of this for us.

The next thing we do is generate a memory layout for the particular size and alignment. This can fail and return an error if the alignment is bad (zero or not a power of two) or if size is too big, otherwise this should succeed. Given our layout,

we can then proceed to actually allocating memory. If the resulting size of the layout is not positive then we don't need to allocate anything (and if fact calling alloc with a zero sized Layout could lead to undefined behavior depending on the architecture). In this case we just cast the alignment to the right type and return it as that is about all we can do.

Otherwise we have a real region of memory we need to allocate so we use the `alloc` function provided by the standard library. It is possible to customize the allocator used, but by default a standard one is used per platform. This is how the interaction with the allocator is exposed regardless. We get back a pointer from `alloc` which is the location of our newly allocated region of memory of the size and alignment specified by our layout. We only return this pointer if it is not null. A null pointer returned from `alloc` most likely means you are out of memory.

If we make it to the end of this function without having already returned something it means either the Layout failed to build or we are probably out of memory. There are a variety of things you could do in this scenario, calling panic is a reasonable one.

> If you use any method that can panic in your Rust code, even if you definitely never panic, your Wasm module will increase quite a bit in size because of extra code related to panics. There are non-panicing alternatives to a lot of methods and there are other things you can do in these scenarios. It is possible to configure your code so that you are not allowed to panic, notably by using `no_std` which means disallowing anything from the `std` module, but that can be extreme (although necessary in some environments).

Okay, we can allocate memory in the Wasm address space via an exposed function in Rust, let's take care of deallocating memory:

src/lib.rs

```
#[no_mangle]
pub unsafe extern "C" fn __free(ptr: *mut u8, size: usize) {
    if size == 0 {
        return
    }
    let align = mem::align_of::<usize>();
    let layout = Layout::from_size_align_unchecked(size, align);
    dealloc(ptr, layout);
}
```

We take a pointer to the memory region we want to deallocate and the size of that region. If the size is zero then there is nothing to do. Otherwise we do the reverse of allocation, we get an alignment, use that to get a Layout, and then pass the pointer and the layout to `dealloc`. Note that our entire function is marked as `unsafe` as part of the signature rather than putting an unsafe block inside the function.

You can either expose a "safe" function which might do unsafe things internally, or you can say calling a function is inherently unsafe. The distinction is meant to imply who is responsible for the invariants. If you expose a safe function then you are responsible for making your invariants clear and usually handling the cases where they are not upheld. If you mark a function as unsafe, you are still responsible for making your invariants clear but you are saying that it is the responsibility of the caller to handle the bad cases. The exact lines are sometimes blurry. Should our greet wrapper function be unsafe rather than just using unsafe internally? Possibly, but we theoretically control the JavaScript code that is calling it and therefore think we can maintain the invariants.

Lastly, we need to implement `__boxed_str_free` to prevent leaking the String we return:

src/lib.rs

```
#[no_mangle]
pub unsafe extern "C" fn __boxed_str_free(ptr: *mut String) {
    let _b = Box::from_raw(ptr);
}
```

We mark this function as unsafe for the same reason as `__free`. `Box::from_raw` is an unsafe function which takes a raw pointer and constructs a box from it. By creating this box and putting it into a local variable we ensure that the `Box` will be dropped at the end of the function body. When the Box is dropped, as it is the sole owner of the String, the String will also be dropped. The input type being `*mut String` is sufficient to tell Rust the right code to execute to drop both the Box and the String as this drives the type inference of `from_raw`.

With all of that out of the way, we can add the code to exercise our Wasm code from JavaScript:

www/index.html

```
    const result = greet("Rust");
    console.log(result);
```

The full code listing for the JavaScript side is here which includes instantiating the Wasm module:

www/index.html

```
<!DOCTYPE html>
<script type="module">
  async function init() {
    const { instance } = await WebAssembly.instantiateStreaming(
      fetch("./hello_raw.wasm")
    );
    const wasm = instance.exports;

    function greet(arg0) {
      const [ptr0, len0] = passStringToWasm(arg0);
```

```
      try {
        const retptr = wasm.greet(ptr0, len0);
        const mem = new Uint32Array(wasm.memory.buffer);
        const rustptr = mem[retptr / 4];
        const rustlen = mem[retptr / 4 + 1];
        const realRet = getStringFromWasm(rustptr, rustlen).slice();
        wasm.__boxed_str_free(retptr);
        return realRet;
      } finally {
        wasm.__free(ptr0, len0);
      }
    }

    function passStringToWasm(arg) {
      const buf = new TextEncoder('utf-8').encode(arg);
      const len = buf.length;
      const ptr = wasm.__malloc(len);
      let array = new Uint8Array(wasm.memory.buffer);
      array.set(buf, ptr);
      return [ptr, len];
    }

    function getStringFromWasm(ptr, len) {
      const mem = new Uint8Array(wasm.memory.buffer);
      const slice = mem.slice(ptr, ptr + len);
      const ret = new TextDecoder('utf-8').decode(slice);
      return ret;
    }

    const result = greet("Rust");
    console.log(result);
  }

  init();
</script>
```

Similarly to the last section, we create a build script to handle creating the Wasm module:

build.sh

```
#!/bin/bash

WABT_BIN=$HOME/Code/wabt/bin
BINARYEN_BIN=$HOME/Code/binaryen/bin
TARGET=wasm32-unknown-unknown
NAME=hello_raw
BINARY=target/$TARGET/release/$NAME.wasm

cargo build --target $TARGET --release
$WABT_BIN/wasm-strip $BINARY
mkdir -p www
$BINARYEN_BIN/wasm-opt -o www/$NAME.wasm -Oz $BINARY
```

Again we make it executable:

```
chmod +x build.sh
```

We also need a server so we repeat the same Python server for convenience:

serve.py

```
#!/usr/bin/env python3

import http.server
import socketserver

PORT = 8080

Handler = http.server.SimpleHTTPRequestHandler

Handler.extensions_map[".wasm"] = "application/wasm"

```

```
12 httpd = socketserver.TCPServer(("", PORT), Handler)
13
14 print("serving at port", PORT)
15 httpd.serve_forever()
```

We want to make it executable so that we can run it directly:

```
chmod +x serve.py
```

Run the server:

```
./serve.py
```

Navigate to `http://localhost:8080/www` and look in the console to see:

```
1 Hello, Rust!
```

The Real Way to Write Wasm

Rust has been at the forefront of the development of WebAssembly and therefore there is a solid set of tools available for automating much of the tedious glue that we saw is necessary for working with complex types. The majority of the heavy lifting is done by the wasm-bindgen[41] crate and CLI. Built on top of wasm-bindgen is the wasm-pack[42] tool. This is a build tool which automates the process of exposing your Wasm module as an NPM module. The wasm-pack documentation has examples and templates for getting up and running with Wasm in the browser and in NodeJS environments. We are going to replicate our simple greet function but this time rely on these tools to do the work.

The first step is to install the wasm-pack[43] CLI. This will be needed later on but is good to get out of the way because you may hit snags based on what is on your system. Currently, wasm-pack uses npm under the hood to handle certain tasks so you will need it installed as well.

Let's create a Rust library to hold our Wasm code:

[41] https://github.com/rustwasm/wasm-bindgen
[42] https://rustwasm.github.io/docs/wasm-pack/introduction.html
[43] https://rustwasm.github.io/wasm-pack/installer/

```
cargo new --lib hello-bindgen
```

As before we need to set our crate type to be a dynamic library, but now we are also going to be adding a dependency on the wasm-bindgen crate:

Cargo.toml

```
[package]
name = "hello-bindgen"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[lib]
crate-type = ["cdylib"]

[dependencies]
wasm-bindgen="^0.2"
```

Next we are going to write the greet function as before but this time we are going to use wasm-bindgen to help:

src/lib.rs

```
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}
```

We import the items in the wasm-bindgen prelude so that the generated code has what it needs to work with. Then getting your code exposed to JavaScript is as simple as adding the `wasm_bindgen` attribute. Our greet function needs to be public to be exposed, but otherwise this is all the code we need to write.

Now we can use wasm-pack to build a JavaScript package that contains our compiled Wasm code with one simple command:

```
wasm-pack build
```

This will produce output in the `pkg` directory:

```
hello-bindgen/pkg/
├── hello_bindgen.d.ts
├── hello_bindgen.js
├── hello_bindgen_bg.d.ts
├── hello_bindgen_bg.wasm
└── package.json
```

This is a JavaScript package which can be used like any other but internally it uses Wasm. The easiest way to see that our function was exported as expected is to look at the generated TypeScript definition file:

pkg/hello_bindgen.d.ts

```
/* tslint:disable */
/**
* @param {string} name
* @returns {string}
*/
export function greet(name: string): string;
```

Now we need to have a JavaScript project that uses our package. We can use npm to create an application based on the wasm-app template:

```
npm init wasm-app hello-bindgen-app
```

This step is unnecessary if you are using the Wasm package in an existing JavaScript application, but for our purposes this is the most expedient path. We modify the package.json file to make our Wasm package a dependency:

hello-bindgen-app/package.json

```
  "dependencies": {
    "hello-bindgen": "file:../pkg"
  },
  "devDependencies": {
    "webpack": "^4.29.3",
    "webpack-cli": "^3.1.0",
    "webpack-dev-server": "^3.1.5",
    "copy-webpack-plugin": "^5.0.0"
  }
```

Then we edit index.js to load our package and call the greet function:

hello-bindgen-app/index.js

```
import * as wasm from "hello-bindgen";

let result = wasm.greet("Rust");
console.log(result);
```

We need to use npm to install the dependencies needed to package and serve our code:

```
npm install
```

Finally, we can use the bundled webpack dev server to view our code by running the start script that comes with the template we used:

```
npm start
```

Again we can navigate to `http://localhost:8080` and look in the console to see:

```
Hello, Rust!
```

What just happened?

The wasm-bindgen crate generates some Rust code based on where you put the attribute. We annotated a function so the crate will generate a wrapper function that handles marshalling the complex data types, `&str` and `String` in this case, into integers that Wasm can handle. Note that we did not say `extern "C"` on our greet function which is by design. As in our manual example, the actual greet function that is exposed ends up being a wrapper which calls into our original greet function which will get a mangled name.

You can put the `wasm_bindgen` attribute on structs, `impl` blocks, and a variety of other Rust items to expose them to JavaScript. The wasm-bindgen docs[44] are a great source for understanding what and how things are exposed and what options you have.

The next step is that wasm-pack uses the wasm-bindgen CLI to generate JavaScript wrapper code based on items annotated with the `wasm_bindgen` attribute. The JavaScript glue code requires you to run an extra step outside of the normal Rust build process. The internals of the code generated by wasm-bindgen on the Rust side means that you really want to use wasm-bindgen to generate the JavaScript wrappers for you as well.

All of the generated code, either in Rust via the attribute, or in JavaScript via the CLI, is quite similar to what we did by hand in the previous section. For example, check out the generated JavaScript wrapper in `pkg/hello_bindgen.js`:

pkg/hello_bindgen.js

```
/**
* @param {string} name
* @returns {string}
*/
export function greet(name) {
    const ptr0 = passStringToWasm(name);
    const len0 = WASM_VECTOR_LEN;
    const retptr = globalArgumentPtr();
    try {
        wasm.greet(retptr, ptr0, len0);
```

[44] https://rustwasm.github.io/docs/wasm-bindgen/

```
        const mem = getUint32Memory();
        const rustptr = mem[retptr / 4];
        const rustlen = mem[retptr / 4 + 1];

        const realRet = getStringFromWasm(rustptr, rustlen).slice();
        wasm.__wbindgen_free(rustptr, rustlen * 1);
        return realRet;


    } finally {
        wasm.__wbindgen_free(ptr0, len0 * 1);

    }

}
```

This should look very familiar. Note that wasm-bindgen uses a slightly more complicated wrapper on the Rust side which requires a different calling convention in Wasm. In particular, note that the greet function takes a return pointer as the first argument and does not return anything. However both approaches achieve the same goal.

Other Wasm Topics

WebAssembly is a broad topic with an increasing surface area. We will try to cover at a high level many of the common questions that arise when people first encounter Wasm.

The DOM and friends

As part of the wasm-bindgen project, there is the web_sys[45] crate which exposes a large number of raw Web APIs. Manipulating the DOM is one thing that is possible using this crate.

[45] https://rustwasm.github.io/wasm-bindgen/api/web_sys/

Threads

There is a proposal[46] to add threads to WebAssembly. As part of this proposal there are also ideas about atomics and other concepts necessary to make shared linear memory across threads manageable. There is a good summary blog post[47] by Alex Crichton which breaks down a lot of the exciting details.

WebAssembly System Interface (WASI)

WebAssembly derives its security features from only having access to what it is explicitly given. That is, Wasm can be compiled to machine code but it cannot do things like open arbitrary sockets unless the system explicitly passes an instance the capability to do so. Today this is done via the JavaScript APIs available to Wasm code running in the browser. But there is an effort to run Wasm outside of the browser.

Outside the browser there is a real system to deal with. This leads to the question of portability of Wasm code across different systems that might want to execute the code. It also begs the question about how to handle safely support system calls like `open` for opening a file.

Wasm is an assembly language for a logical machine which can therefore be implemented virtually on any number of platforms. The WebAssembly System Interface, or WASI, is an attempt to standardize the system calls that Wasm knows about so that different implementations can build to a spec and therefore abstract the underlying operating system from the assembly language. This is being designed with portability and security as the paramount concerns. It is still in early days but it is a very exciting proposition.

[46] https://github.com/WebAssembly/threads
[47] https://rustwasm.github.io/2018/10/24/multithreading-rust-and-wasm.html

Command Line Applications

Many command line applications that we use every day were written long ago when the dominate high level language was C. Ever since there have been improvements made to these fundamental programs as well as attempts to replace them. One niche where Rust fits nicely is in building these command line applications.

One example is ripgrep[48] which can replace grep for many use cases, is faster, and provides a lot of nice extra features. Rust is a big part of the success story of `ripgrep`.

One feature that is becoming increasingly more important is the ability to support multiple platforms without the need for parallel codebases or ugly hacks. Mac and Linux support have gotten much closer over the years, but Windows is still a much harder target to hit for a wide variety of languages. This is one area that Go has worked quite well at addressing and Rust also shines here as well.

Building command line utilities can greatly simplify your life if you spend a lot of time in the terminal. Sometimes stitching together awk, Perl, and Bash for the third time starts to make you wonder if there is a better way. Before Rust's ecosystem got to the current state it might have made sense to reach for Python or Ruby with their plethora of libraries to build a simple command line tool. But we are now at a point where you can have performance, safety, and a ton of great features by putting together crates out in the wild.

Initial setup

Let's walk through building a simple command line application for making HTTP requests. This has sufficiently complexity to be interesting, but not too much so that we can focus on the Rust parts of the problem rather than the details of this particular application.

We are going to build something like `cURL` with a few extra features, so let's create a new project:

[48] https://blog.burntsushi.net/ripgrep/

```
cargo new hurl
```

Making an MVP

We are going to walk through quite a bit of code before we get to a runnable version of our application. This is just to get us to an MVP with a few simple features. After that we will expand our feature set with some nice things that curl is missing.

Basic feature set

We want to support some basic use cases. In all scenarios we want to:

- Parse the response if it is JSON
- Print out information about the request including headers
- Print out the response as nicely as possible

Let's go through a few simple motivating examples:

- Make a GET request to `example.com`

```
$ hurl example.com
```

Here we see that the default is to make a GET request as no other method is given and no data is being sent.

- Make a POST request to example.com with a JSON object `{"foo":"bar"}` as the data:

```
$ hurl example.com foo=bar
```

Here we see the default when data is being sent is to make a POST request. Also, the default is to make a request using JSON rather than a form.

- Make a PUT request to example.com using HTTPS, set the `X-API-TOKEN` header to `abc123`, and upload a file named `foo.txt` with the form field name `info`:

```
$ hurl -s -f PUT example.com X-API-TOKEN:abc123 info@foo.txt
```

We see the `-s` for `--secure` option used to turn on HTTPS and `-f` option used to turn on sending as a form. We want headers to be specified with a colon separating the name from the value and files to be uploaded by using `@` between the form field name and the path to the file on disk.

- Make a form POST request to example.com, use `bob` as the username for basic authentication, set the query parameter of `foo` equal to `bar`, and set the form field `option` equal to `all`:

```
$ hurl -f -a bob POST example.com foo==bar option=all
```

In this last example, before making the request, the user should be prompted for their password which should be used for the basic authentication scheme.

Building the features

First things first, we need to add dependencies to our manifest:

Cargo.toml

```
[package]
name = "hurl"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
structopt = "0.3"
heck = "0.3"
log = "0.4"
pretty_env_logger = "0.3"
serde = "1.0"
serde_json = "1.0"
reqwest = "0.9.20"
rpassword = "4.0"
```

Whoa that is a lot of dependencies. It is easiest to get most of these out of the way now as working them in after the fact makes things overly complicated. We will get to each of these in detail as we use them, but the quick overview of each is:

- `structopt` - command line argument parsing, and much more.
- `heck` - converting strings to different cases (e.g. snake_case, camelCase).
- `log` - logging facade for outputting verbose information.
- `pretty_env_logger` - logging implementation that works with `log`.
- `serde` - (de)serialization of data.
- `serde_json` - `serde` for JSON data.
- `reqwest` - HTTP client.
- `rpassword` - ask a user for a password without echoing it to the terminal.

We are going to walk through building this application by going file by file as we build out the necessary bits.

The main entry point: main.rs

As we are building a binary application, we will need some code in `main.rs` to kick things off. As the functionality here will not be used in a library, and is in fact mostly building on other libraries, we will forgo the advice of creating a separate library that we call into and instead work directly in `main.rs`.

Let's bring in some imports to get started:

src/main.rs

```
1 use structopt::StructOpt;
2 use heck::TitleCase;
3 use log::trace;
```

The `structopt` crate defines a trait `StructOpt` and a custom derive which allows you to derive that trait for a type you create. These two pieces together create a system for declaratively specifying how your application takes input from the command line.

Underlying `structopt` is another crate called `clap`. You can, and many people do, build a command line application directly by interacting with the `clap` crate.

However, `structopt` abstracts a lot of those details away, as it is mostly boilerplate, and instead allows you to focus on describing your options using the type system.

If this sounds too abstract at the moment, bear with me, as it will all become clear shortly. Nonetheless, we need to import the trait as we will use that functionality in our main function.

We import `TitleCase` which is also a trait from the `heck` crate so that we can convert strings into their title cased equivalents. For example, `this_string` would be converted to `ThisString`. We could write this logic ourselves, but why bother to worry about all the edge cases when a high quality crate already exists for this purpose.

Finally, the import of the `trace` macro from the `log` crate is to enable us to leverage the `log` crate's nice features to implement a form of verbose logging. Logging in Rust has largely concentrated on the abstraction provided by the `log` crate.

When you write a library or other tool that wants to generate logs, you use the macros in the `log` crate to write those messages at various verbosity levels. When you are creating an application where you want to enable seeing these different log messages, you bring in a separate crate (or write some extra code yourself) which provides the concrete implementation of how those macros get turned into output. It is possible to have an implementation that writes everything to a file, or that only turns on some verbosity levels, or that writes to the terminal with pretty colors.

As we are creating a binary, we depend on a concrete implementation of the logging functionality, but for actually generating the messages we just need the macros defined in the `log` crate.

As part of building this application, we are going to split up the functionality into different modules. We define those modules here that we will soon write so they become part of our application:

src/main.rs

```
5 mod app;
6 mod client;
7 mod errors;
```

As you should recall, this tells the compiler to look for files (or directories) with those names and to insert that code here with the appropriate scoping.

We next use a `use` statement to bring our to be written error type into scope to make our type signatures easier to write:

src/main.rs

```
9  use errors::HurlResult;
```

One feature we want to support is pretty printing JSON responses. An aspect of pretty printing that can be quite handy is making sure that the keys are sorted. JSON objects do not have a well-defined order so we are free to print them anyway we want. There is an argument that we should faithfully represent the result based on the order of the bytes from the server. However, we are writing our own tool and are therefore free to make decisions such as this. Thus, let's create a type alias that we will be able to use with serde to get an ordered JSON object:

src/main.rs

```
11 type OrderedJson = std::collections::BTreeMap<String, serde_json::Value\
12 >;
```

Rust has multiple hash map implementations in the standard library, one in particular is the `BTreeMap` which stores entires sorted by the key. In this case, we expect our response to have string keys and arbitrary JSON values. This type alias is not doing any work, but we will see how we use it with serde to get the result we are looking for.

We are going to practice a little speculative programming here by writing a main function with code for things that do not yet exist. Given that structure, we then just have to make those things exist and we will be done. Without further ado, our main function:

src/main.rs

```
fn main() -> HurlResult<()> {
    let mut app = app::App::from_args();
    app.validate()?;

    if let Some(level) = app.log_level() {
        std::env::set_var("RUST_LOG", format!("hurl={}", level));
        pretty_env_logger::init();
    }

    match app.cmd {
        Some(ref method) => {
            let resp = client::perform_method(&app, method)?;
            handle_response(resp)
        }
        None => {
            let url = app.url.take().unwrap();
            let has_data = app.parameters.iter().any(|p| p.is_data());
            let method = if has_data {
                reqwest::Method::POST
            } else {
                reqwest::Method::GET
            };
            let resp = client::perform(&app, method, &url, &app.paramet\
ers)?;
            handle_response(resp)
        }
    }
}
```

First, we return a Result, in particular our custom `HurlResult` with a success type of `()` meaning we only have a meaningful value to report for errors, and that the `Ok` case just means everything worked. This shows that the `HurlResult` type we need to write is generic over the success type as we have seen other result type aliases before.

The first line of our function uses a call to `from_args` which is defined on the `StructOpt` trait. Therefore, we need a struct called `App` in the `app` module which implements this trait. This does all of the command line argument parsing including exiting if something can't be parsed as well as printing a help message when necessary.

We then have a call to `validate` which uses the `?` operator to exit early if this function fails. This is not part of the argument parsing that `StructOpt` does. Rather, this is for handling certain constraints on the arguments that `StructOpt` is unable to enforce. We will see what this is when we get to defining `App`, but it just shows that sometimes we need workarounds even if a crate does almost everything we need.

There are two further sections. The first uses the `log_level` method on our app to get a value to setup logging. The `pretty_env_logger` crate uses environment variables to configure what to print, so we explicitly set the `RUST_LOG` environment variable based on the value we get. The format is `RUST_LOG=binary_name=level` where `binary_name` is not surprisingly the name of your binary and `level` is one of the five level values that `log` defines: `trace`, `debug`, `info`, `warn`, and `error`. After setting the environment variable, we call `pretty_env_logger::init()` to actually hook this logging implementation into our use of the macros from the `log` crate.

The second piece is the heart of our application. We use the `cmd` (short for command), property on our app to direct what type of request to make. There are two cases, either we got a command which specifies the HTTP verb to use, in that case we use the client module to make the request and then call a `handle_response` function with the result.

If we did not get a command, i.e. `app.cmd` matches `None`, then we are in the default case where we just got a URL. In this case, we make a GET request if we do not have any data arguments, otherwise we make a POST request. We also call a method on the client module to make this request and pipe through to the same `handle_response` function.

So in either case we end up with a response from our client module that we need to handle. Let's turn to that handler function:

src/main.rs

```
fn handle_response(
    mut resp: reqwest::Response,
) -> HurlResult<()> {
```

First, the signature. We expect a response as input, which in this case is just the `Response` type from the `reqwest` crate, and we return our result type.

The purpose of our tool is to print useful information to standard output and therefore "handling" the response means doing that printing. First, we are going to build up a string based on the metadata about the response. Then we will turn to printing the body.

Our first useful piece of data is the status code of the response as well as the version of HTTP that was used to communicate said response.

src/main.rs

```
    let status = resp.status();
    let mut s = format!(
        "{:?} {} {}\n",
        resp.version(),
        status.as_u16(),
        status.canonical_reason().unwrap_or("Unknown")
    );
```

The string `s` is going to be our buffer that we use for building up the output. We start out by using `format!` to create a `String` based on the version stored in the response, the status code, and the description of that status. Example results for this string are "HTTP/1.1 200 OK" or "HTTP/1.1 403 Forbidden". We use the `Debug` output of `resp.version()` because that is defined to be what we want to show. Sometimes `Display` is not defined or `Debug` is what you want to see so don't always think of `{:?}` as strictly for debugging.

Next up we want to show the headers from the response. We are going to gather them into a vector which we will combine into a string after the fact.

src/main.rs

```
51      let mut headers = Vec::new();
52      for (key, value) in resp.headers().iter() {
53          let nice_key = key.as_str().to_title_case().replace(' ', "-");
54          headers.push(format!(
55              "{}: {}",
56              nice_key,
57              value.to_str().unwrap_or("BAD HEADER VALUE")
58          ));
59      }
```

The response type has a `headers` function which returns a reference to a `Header` type that gives us access to the headers. We have to explicitly turn it into an iterator by calling the `iter` so that we can process each key/value pair. We are again using the `format!` macro to construct a string for each header.

We create a nice to read key by transforming the raw key in the header map into a consistent style. The `to_title_case` method is available because we imported the `TitleCase` trait from `heck`.

We expect the value to have a string representation, but in case that fails we use a quick and dirty "BAD HEADER VALUE" replacement. We do not want to fail the entire response printing if one header value cannot be printed correctly. Luckily we can use the `unwrap_or` method to easily handle this case.

One special exception is content length. The reqwest crate does not treat content length as a normal header value and instead provides a `content_length` function on the response type to get this value. Let's use this function to get a content length to add to your list of headers:

src/main.rs

```
60      let result = resp.text()?;
61      let content_length = match resp.content_length() {
62          Some(len) => len,
63          None => result.len() as u64,
64      };
65      headers.push(format!("Content-Length: {}", content_length));
```

There is one bit of complication, this function returns an `Option` and thus we have to deal with computing a value if this function returns `None`. We accomplish this by getting the `text` of the response and computing a length.

As far as I can tell, reqwest behaves this way because of compression. The content length of the response is not necessarily the same as the content length that is given in the response header because the actually response body could be compressed. After decompressing the body, we end up with a different length. The library returns `None` in this case to signal that if you want to compute an accurate content length, you have to do it yourself.

This leads to the question of what do you want to show in this case? The length given in the response header or the length of the decompressed body? We choose the length of the decompressed body as that is more inline with a user's expectations.

Now that we have all of our headers, we can put it together with our status string and print our first piece of information.

src/main.rs

```
66      headers.sort();
67      s.push_str(&(&headers[..]).join("\n"));
68      println!("{}", s);
```

We put the headers into a vector so that we can sort by the name of the header here. This makes it easier to find headers in the output. We then use the `join` method on string slices to turn the list of headers into a newline separated string. As `join` is not a method on `Vec`, we have to use `&headers[..]` to get a reference to a slice of type `&[String]`. We then turn the output of that function from `String` to `&str` with the

extra `&` at the beginning. This allows us to pass the result to `push_str` which appends onto our already constructed status string `s`. Finally we print the whole thing out.

Finally, we want to print out the body of the response. We already stored the raw text of the response in the variable `result` which we used for computing content length. One nicety we want to provide is pretty printing JSON results if the result is JSON. Therefore, we try to parse the body into JSON and then pretty print if that works, otherwise we just print the raw body.

src/main.rs

```
70      let result_json: serde_json::Result<OrderedJson> = serde_json::from\
71  _str(&result);
72      match result_json {
73          Ok(result_value) => {
74              let result_str = serde_json::to_string_pretty(&result_value\
75  )?;
76              println!("{}", result_str);
77          }
78          Err(e) => {
79              trace!("Failed to parse result to JSON: {}", e);
80              println!("{}", result);
81          }
82      }
83
84      Ok(())
85  }
```

Note here that as we are specifying the type of `result_json` as `serde_json::Result<OrderedJson>` the actually representation of the JSON will be ordered by the keys if it is correctly parsed. This is because of our type alias for `OrderedJson` telling serde that it should use a BTreeMap as the container for the top level JSON object.

The application module

We already wrote code that depends on the existence of an `app` module, so let's make that real by creating `src/app.rs` and getting started with some imports:

src/app.rs

```
use log::{debug, trace};
use std::convert::TryFrom;
use structopt::StructOpt;

use crate::errors::{Error, HurlResult};
```

This module is the core of how we interact with the user via command line arguments. The `App` struct is both the data that configures what we will do as well as the majority of the code we need to write to handle getting that data. This is because of the `StructOpt` custom derive macro which we take advantage of.

The `structopt` crate is built on the `clap` crate. You may also see people using `Clap` to create command line applications. We are doing the same thing, but writing significantly less code by using a form of declarative programming. We are going to build up a large struct definition in pieces and explain as we go, but in the end we will have almost everything we need to build a full featured command line application.

Our first step is to declare our struct with the name `App`:

src/app.rs

```
/// A command line HTTP client
#[derive(StructOpt, Debug)]
#[structopt(name = "hurl")]
pub struct App {
```

Comments that start with three slashes, `///`, are known as doc comments. They are treated specially by the compiler and used for auto-generating crate documentation. Notice that we put such a comment on top of the derives for our struct which will associate that comment with the struct as a whole.

We derive `Debug` which is normal and we have seen before, but the magic happens by deriving `StructOpt`. This is a type of macro known as a custom derive which means that code in the `structopt` crate will be given our struct definition as input and will output code which will then be included in our crate. This is how the `Debug` derive works as well except that custom code is part of the standard library. Usually these

derives are used to implement traits, but they can also be used for any number of other purposes.

We also see an attribute `#[structopt(name = "hurl")]` which is ignored by the rest of the compiler but is something that the `StructOpt` derive uses for customization. Doc comments are included as part of the struct definition to the custom derive and therefore `structopt` uses the doc comment on this struct as part of the help message which gets created as part of the code that is generated.

We then create several options for our application to accept at the command line by creating fields on our struct:

src/app.rs

```
11      /// Activate quiet mode.
12      ///
13      /// This overrides any verbose settings.
14      #[structopt(short, long)]
15      pub quiet: bool,
16
17      /// Verbose mode (-v, -vv, -vvv, etc.).
18      #[structopt(short, long, parse(from_occurrences))]
19      pub verbose: u8,
20
21      /// Form mode.
22      #[structopt(short, long)]
23      pub form: bool,
24
25      /// Basic authentication.
26      ///
27      /// A string of the form `username:password`. If only
28      /// `username` is given then you will be prompted
29      /// for a password. If you wish to use no password
30      /// then use the form `username:`.
31      #[structopt(short, long)]
32      pub auth: Option<String>,
33
34      /// Bearer token authentication.
```

```
35      ///
36      /// A token which will be sent as "Bearaer <token>" in
37      /// the authorization header.
38      #[structopt(short, long)]
39      pub token: Option<String>,
40
41      /// Default transport.
42      ///
43      /// If a URL is given without a transport, i.e example.com/foo
44      /// http will be used as the transport by default. If this flag
45      /// is set then https will be used instead.
46      #[structopt(short, long)]
47      pub secure: bool,
48
49      /// The HTTP Method to use, one of: HEAD, GET, POST, PUT, PATCH, DE\
50  LETE.
51      #[structopt(subcommand)]
52      pub cmd: Option<Method>,
53
54      /// The URL to issue a request to if a method subcommand is not spe\
55  cified.
56      pub url: Option<String>,
```

The types of the variables determine what type of flag/option is created. For example, a field with type `bool` like `quiet` creates a flag which sets the field to true if it is present and is otherwise set to false. Each field has documentation in the help message created by the doc comments. The attributes on the fields are further used for customization. For example, `short, long` says that the field should be usable using a short form like `-q` for `quiet` and a long form `--quiet`.

The structopt docs explain exactly what each type implies but they are mostly what one would expect. An `Option` around a type signifies that argument is optional, whereas a non-optional field that is not a `bool` would be treated as a required positional argument. We use `parse(from_occurrences)` on the `verbose` field to allow the `u8` field to be inferred from the number of times the argument is passed. This is a common pattern to use for verbosity by command line tools.

The one special field you might notice is `cmd` which has a attribute of `subcommand`. We will get to this below, but basically this means that we need to define a `Method` type which also derives `StructOpt` which will be used to create a subprogram here. As this is wrapped in an `Option` this is not required.

Finally, the last piece of data that we accept is a list of parameters:

src/app.rs

```
56      /// The parameters for the request if a method subcommand is not sp\
57  ecified.
58      ///
59      /// There are seven types of parameters that can be added to a comm\
60  and-line.
61      /// Each type of parameter is distinguished by the unique separator\
62   between
63      /// the key and value.
64      ///
65      /// Header -- key:value
66      ///
67      ///   e.g. X-API-TOKEN:abc123
68      ///
69      /// File upload -- key@filename
70      ///
71      ///   this simulates a file upload via multipart/form-data and requ\
72  ires --form
73      ///
74      /// Query parameter -- key==value
75      ///
76      ///   e.g. foo==bar becomes example.com?foo=bar
77      ///
78      /// Data field -- key=value
79      ///
80      ///   e.g. foo=bar becomes {"foo":"bar"} for JSON or form encoded
81      ///
82      /// Data field from file -- key=@filename
83      ///
84      ///   e.g. foo=@bar.txt becomes {"foo":"the contents of bar.txt"} o\
```

```
r form encoded
    ///
    /// Raw JSON data where the value should be parsed to JSON first --\
 key:=value
    ///
    ///   e.g. foo:=[1,2,3] becomes {"foo":[1,2,3]}
    ///
    /// Raw JSON data from file -- key:=@filename
    ///
    ///   e.g. foo:=@bar.json becomes {"foo":{"bar":"this is from bar.j\
son"}}
    #[structopt(parse(try_from_str = parse_param))]
    pub parameters: Vec<Parameter>,
}
```

This is mostly documentation, and that documentation should explain what is going on. The `Vec<Parameter>` type means that we accept multiple values of this type of input. As this is a custom type we also need to implement `parse_param` to work with the attribute that allows us to define a custom parsing function.

With our definition out of the way, most of the hard work of creating a nice command line interface is taken care of. We have a few pieces still left to take care of to handle some of the custom pieces of our app, but for most applications just defining the struct with the right field types can be all you need to do.

We need to add some methods to our `App`. There is no way to declaratively say that we want exactly one of `cmd` or `url` to exist. This is partially because of the way we define `Method` as we will see below and it is partially due to limitations in the custom derive. Therefore, we create a `validate` method to check this constraint after the parsing has completed and will allow us to error out in that case.

src/app.rs

```
impl App {
    pub fn validate(&mut self) -> HurlResult<()> {
        if self.cmd.is_none() && self.url.is_none() {
            return Err(Error::MissingUrlAndCommand);
        }
        Ok(())
    }

    pub fn log_level(&self) -> Option<&'static str> {
        if self.quiet || self.verbose <= 0 {
            return None;
        }

        match self.verbose {
            1 => Some("error"),
            2 => Some("warn"),
            3 => Some("info"),
            4 => Some("debug"),
            _ => Some("trace"),
        }
    }
}
```

The other thing we add is a helper to turn the `quiet` and `verbose` settings into a string log level for use with our logging implementation.

Now we can turn to our data structure for the subcommand:

src/app.rs

```
#[derive(StructOpt, Debug)]
#[structopt(rename_all = "screaming_snake_case")]
pub enum Method {
    HEAD(MethodData),
    GET(MethodData),
    PUT(MethodData),
    POST(MethodData),
    PATCH(MethodData),
    DELETE(MethodData),
}
```

We create an `enum` because we want to use the name of the enum, which is an HTTP method, as the name of the subcommand. `StructOpt` works just as well with an enum as a struct, provided you follow the rules for what the variants can be. In our case, each variant has the same inner data which itself derives `StructOpt`. The one extra attribute we use here `rename_all = "screaming_snake_case"` is so that our program uses the form `hurl POST whatever.com` instead of `hurl post whatever.com`. This is a purely stylistic choice.

The inner data for each enum variant is a struct to contain the URL and the parameters:

src/app.rs

```
#[derive(StructOpt, Debug)]
pub struct MethodData {
    /// The URL to request.
    pub url: String,

    /// The headers, data, and query parameters to add to the request.
```

All of the doc comments on the `parameters` field in the top level `App` struct are repeated here in the code, but we omit them for brevity:

src/app.rs

```
191     #[structopt(parse(try_from_str = parse_param))]
192     pub parameters: Vec<Parameter>,
193 }
```

Generally, the same thing repeated can be a sign of some missing abstraction or constant, but in this case the repeated documentation is about as good as we can do.

We also define one helper method on our `Method` enum to get the data out of each variant:

src/app.rs

```
127 impl Method {
128     pub fn data(&self) -> &MethodData {
129         use Method::*;
130         match self {
131             HEAD(x) => x,
132             GET(x) => x,
133             PUT(x) => x,
134             POST(x) => x,
135             PATCH(x) => x,
136             DELETE(x) => x,
137         }
138     }
139 }
```

We have been using the `Parameter` type in our definitions above, so it is now time to define it:

src/app.rs

```
#[derive(Debug)]
pub enum Parameter {
    // :
    Header { key: String, value: String },
    // =
    Data { key: String, value: String },
    // :=
    RawJsonData { key: String, value: String },
    // ==
    Query { key: String, value: String },
    // @
    FormFile { key: String, filename: String },
    // =@
    DataFile { key: String, filename: String },
    // :=@
    RawJsonDataFile { key: String, filename: String },
}
```

This enum is used to specify the data for the different types of parameters that accept. Each one has a particular use case and having the nice shorthand is quite helpful when mixing and matching.

Our final task in this module is to write our `parse_param` function which will take a string and maybe turn it into a `Parameter`. In order to make this task easier, we define a `Token` type:

src/app.rs

```
#[derive(Debug)]
enum Token<'a> {
    Text(&'a str),
    Escape(char),
}
```

We have these special separator characters like : and ==, but we need some notion

of escaping to allow those to appear in keys and values. We further write a helper function to take string and parse it into a vector of these tokens:

src/app.rs

```
fn gather_escapes<'a>(src: &'a str) -> Vec<Token<'a>> {
    let mut tokens = Vec::new();
    let mut start = 0;
    let mut end = 0;
    let mut chars = src.chars();
    loop {
        let a = chars.next();
        if a.is_none() {
            if start != end {
                tokens.push(Token::Text(&src[start..end]));
            }
            return tokens;
        }
        let c = a.unwrap();
        if c != '\\' {
            end += 1;
            continue;
        }
        let b = chars.next();
        if b.is_none() {
            tokens.push(Token::Text(&src[start..end + 1]));
            return tokens;
        }
        let c = b.unwrap();
        match c {
            '\\' | '=' | '@' | ':' => {
                if start != end {
                    tokens.push(Token::Text(&src[start..end]));
                }
                tokens.push(Token::Escape(c));
                end += 2;
                start = end;
```

```
296                 }
297                 _ => end += 2,
298             }
299         }
300 }
```

This is not the most exciting or elegant parsing code. We use a pair of indexes into the source string along with possibly some lookahead to tokenize the input. Basically this is looking for `\`, `=`, `@`, and `:` following a `\` character to indicate that this otherwise special character should be escaped and treated as a literal. Everything else gets turned into a piece of text.

Finally, we can write our `parse_param` function which gets a string from the command line and tries to turn it into a `Parameter` or an appropriate error:

src/app.rs

```
302 fn parse_param(src: &str) -> HurlResult<Parameter> {
303     debug!("Parsing: {}", src);
304     let separators = [":=@", "=@", "==", ":=", "@", "=", ":"];
305     let tokens = gather_escapes(src);
306 
307     let mut found = Vec::new();
308     let mut idx = 0;
309     for (i, token) in tokens.iter().enumerate() {
310         match token {
311             Token::Text(s) => {
312                 for sep in separators.iter() {
313                     if let Some(n) = s.find(sep) {
314                         found.push((n, sep));
315                     }
316                 }
317                 if !found.is_empty() {
318                     idx = i;
319                     break;
320                 }
321             }
```

```
322             Token::Escape(_) => {}
323         }
324     }
325     if found.is_empty() {
326         return Err(Error::ParameterMissingSeparator(src.to_owned()));
327     }
328     found.sort_by(|(ai, asep), (bi, bsep)| ai.cmp(bi).then(bsep.len().c\
329 mp(&asep.len())));
330     let sep = found.first().unwrap().1;
331     trace!("Found separator: {}", sep);
332 
333     let mut key = String::new();
334     let mut value = String::new();
335     for (i, token) in tokens.iter().enumerate() {
336         if i < idx {
337             match token {
338                 Token::Text(s) => key.push_str(&s),
339                 Token::Escape(c) => {
340                     key.push('\\');
341                     key.push(*c);
342                 }
343             }
344         } else if i > idx {
345             match token {
346                 Token::Text(s) => value.push_str(&s),
347                 Token::Escape(c) => {
348                     value.push('\\');
349                     value.push(*c);
350                 }
351             }
352         } else {
353             if let Token::Text(s) = token {
354                 let parts: Vec<&str> = s.splitn(2, sep).collect();
355                 let k = parts.first().unwrap();
356                 let v = parts.last().unwrap();
357                 key.push_str(k);
```

```
                value.push_str(v);
            } else {
                unreachable!();
            }
        }
    }

    if let Ok(separator) = Separator::try_from(*sep) {
        match separator {
            Separator::At => Ok(Parameter::FormFile {
                key,
                filename: value,
            }),
            Separator::Equal => Ok(Parameter::Data { key, value }),
            Separator::Colon => Ok(Parameter::Header { key, value }),
            Separator::ColonEqual => Ok(Parameter::RawJsonData { key, v\
alue }),
            Separator::EqualEqual => Ok(Parameter::Query { key, value }\
),
            Separator::EqualAt => Ok(Parameter::DataFile {
                key,
                filename: value,
            }),
            Separator::Snail => Ok(Parameter::RawJsonDataFile {
                key,
                filename: value,
            }),
        }
    } else {
        unreachable!();
    }
}
```

We use our `gather_escapes` function to tokenize the input. Then we loop over those tokens looking for separators. We are trying to find the earliest and longest separator

as some of them have overlap. The `found` vector will contain all of the separators that match in the first text segment with an separator. We also store the index in the segment where this separator starts. If we have anything in that list we can stop looking for more, so we break. We error out if no separators are found as that is an erroneous state.

We sort by location and then length of the separator to match our desired condition. Then as we know the vector is not empty we can extract the separator for this parameter from the first element in the vector.

The next chunk of code is using the data we have stored so far to construct a key and value for the particular separator.

Finally, we use the text value of the separator to get a separator type which we then use to construct the appropriate `Parameter`. The code for going from a separator string to a `Separator` as well as the `Separator` enum definition is straightforward:

src/app.rs

```
230 #[derive(Debug)]
231 enum Separator {
232     Colon,
233     Equal,
234     At,
235     ColonEqual,
236     EqualEqual,
237     EqualAt,
238     Snail,
239 }
240
241 impl TryFrom<&str> for Separator {
242     type Error = ();
243
244     fn try_from(value: &str) -> Result<Self, Self::Error> {
245         match value {
246             ":" => Ok(Separator::Colon),
247             "=" => Ok(Separator::Equal),
248             "@" => Ok(Separator::At),
249             ":=" => Ok(Separator::ColonEqual),
```

```
            "==" => Ok(Separator::EqualEqual),
            "=@" => Ok(Separator::EqualAt),
            ":=@" => Ok(Separator::Snail),
            _ => Err(()),
        }
    }
}
```

The client module

We need some mechanism for actually making HTTP requests based on all of this configuration we are building. The module that is responsible for this we call `client`, so let's get started as usual with some imports in `src/client.rs`:

src/client.rs

```
use crate::app::{App, Method, Parameter};
use crate::errors::{Error, HurlResult};
use log::{info, debug, trace, log_enabled, self};
use reqwest::multipart::Form;
use reqwest::{Client, RequestBuilder, Response, Url};
use serde_json::Value;
use std::collections::HashMap;
use std::fs::File;
use std::io::BufReader;
use std::time::Instant;
```

A lot more imports than usual as there is quite a bit to take care of in this module. We are going to build up many small functions to which can be put together to make the different requests we are trying to make. Let's first recall how we call the client module from main:

src/main.rs

```
22      match app.cmd {
23          Some(ref method) => {
24              let resp = client::perform_method(&app, method)?;
25              handle_response(resp)
26          }
27          None => {
28              let url = app.url.take().unwrap();
29              let has_data = app.parameters.iter().any(|p| p.is_data());
30              let method = if has_data {
31                  reqwest::Method::POST
32              } else {
33                  reqwest::Method::GET
34              };
35              let resp = client::perform(&app, method, &url, &app.paramet\
36  ers)?;
37              handle_response(resp)
38          }
39      }
```

We see that there are two entry points, `perform_method` and `perform`. Let's start with `perform_method`:

src/client.rs

```
12  pub fn perform_method(
13      app: &App,
14      method: &Method,
15  ) -> HurlResult<Response> {
16      let method_data = method.data();
17      perform(
18          app,
19          method.into(),
20          &method_data.url,
21          &method_data.parameters,
```

```
    )
}
```

This is a simple helper function which takes our `Method` struct and gets the relevant data from it so that we can call the more general `perform` function. The only missing piece that we rely on is the `method.into()` call which converts our `Method` struct into the `reqwest::Method` type. We need to write the code for that conversion, which we will put into our `app` module alongside the definition of `Method`:

src/app.rs

```
impl From<&Method> for reqwest::Method {
    fn from(m: &Method) -> reqwest::Method {
        match m {
            Method::HEAD(_) => reqwest::Method::HEAD,
            Method::GET(_) => reqwest::Method::GET,
            Method::PUT(_) => reqwest::Method::PUT,
            Method::POST(_) => reqwest::Method::POST,
            Method::PATCH(_) => reqwest::Method::PATCH,
            Method::DELETE(_) => reqwest::Method::DELETE,
        }
    }
}
```

This is a straightforward implementation of the `From` trait which as we have seen before gives us the reciprocal `Into` trait for free which we are using in our function call.

So we just need to implement the general `perform` function and we will be done:

src/client.rs

```
pub fn perform(
    app: &App,
    method: reqwest::Method,
    raw_url: &str,
    parameters: &Vec<Parameter>,
) -> HurlResult<Response> {
```

The signature says that we take in the various configuration data and try to return a response. We chose to call this module `client` because that is the name of the type that reqwest uses for the object that can be used to make HTTP requests. Our first step is therefore to create a `Client` which we imported from reqwest, parse the `url` into a something useful, and then further validate our parameters based on whether or not it is a multipart request:

src/client.rs

```
    let client = Client::new();
    let url = parse(app, raw_url)?;
    debug!("Parsed url: {}", url);

    let is_multipart = parameters.iter().any(|p| p.is_form_file());
    if is_multipart {
        trace!("Making multipart request because form file was given");
        if !app.form {
            return Err(Error::NotFormButHasFormFile);
        }
    }
```

We are using a helper here in the call to determine whether this is a multipart request, `p.is_form_file()` which we need to define on `Parameter`. We again go back to `app.rs` to define some helpers on `Parameter`:

src/app.rs

```
impl Parameter {
    pub fn is_form_file(&self) -> bool {
        match *self {
            Parameter::FormFile { .. } => true,
            _ => false,
        }
    }

    pub fn is_data(&self) -> bool {
        match *self {
            Parameter::Header { .. } => false,
            Parameter::Query { .. } => false,
            _ => true,
        }
    }
}
```

Without these helpers these match statements would be quite messy written inline inside the closure and would leak quite a bit of the implementation details of `Parameter` which is not necessary.

Let's finish up `perform` by building and sending the request:

src/client.rs

```
    let mut builder = client.request(method, url);
    builder = handle_parameters(builder, app.form, is_multipart, parame\
ters)?;
    builder = handle_auth(builder, &app.auth, &app.token)?;

    if log_enabled!(log::Level::Info) {
        let start = Instant::now();
        let result = builder.send().map_err(From::from);
        let elapsed = start.elapsed();
        info!("Elapsed time: {:?}", elapsed);
        result
```

```
54      } else {
55          builder.send().map_err(From::from)
56      }
57  }
```

The `Client` type has a request method which returns a builder given a HTTP method and a URL. We use some soon-to-be written helpers to modify the builder with our various configuration details. Finally, we send the request with `builder.send().map_err(From::from)`. However, we have two cases, one where logging is enabled at at least the `Info` level and when it is not. If we are logging, then we use the time facilities in the standard library to compute the time required to get the response back. If we are not logging, then we just send the request. The `map_err(From::from)` bit is so that we can turn the error possibly returned from reqwest into our custom error type. We will see how this is implemented when we get to the error module.

Our `perform` function is done, but we still have a few speculative helpers left to write before we can call this module quits. First, let's parse the raw URL string we got into the `Url` type exposed by reqwest:

src/client.rs

```
149 fn parse(app: &App, s: &str) -> Result<Url, reqwest::UrlError> {
150     if s.starts_with(":/") {
151         return Url::parse(&format!("http://localhost{}", &s[1..]));
152     } else if s.starts_with(":") {
153         return Url::parse(&format!("http://localhost{}", s));
154     }
155     match Url::parse(s) {
156         Ok(url) => Ok(url),
157         Err(_e) => {
158             if app.secure {
159                 Url::parse(&format!("https://{}", s))
160             } else {
161                 Url::parse(&format!("http://{}", s))
162             }
163         }
164     }
165 }
```

We provide two convenience ways to call localhost URLs. First, if you want to use the default port of 80 on localhost then you can specify the URL as `:/some_path` where `some_path` is optional. In that case we strip off the leading colon and interpolate the rest of the given URL into a string which explicitly mentions localhost.

The other case is if you put something after the color which is not a slash then that is interpreted to mean that you want to specify the localhost port along with your URL so we just use append the raw URL to localhost. In other words, if you want to make a request to say `localhost:8080` you can just use `:8080`. This is a nice little convenience for local development. In both cases we use the `Url::parse` method to do the heavy lifting.

If neither of these two scenarios applies, then we try to parse the given string directly. If that succeeds then we are good to go and just return that. Otherwise, we try to add a scheme to the given URL. The reqwest `Url::parse` function requires a scheme so just using `example.com` would fail to parse. The `App` struct has a `secure` flag for whether to use `https` by default, so we switch on that value to decide which scheme to try. We return the result from this call to `parse` directly as there isn't much else to try if this also fails.

Let's turn to handling the various different parameters we could add to the request. This function is long so at first glance it might be intimidating. However, taken step by step we will see the length is driven by the fact that we support seven different parameter types rather than any lurking complexity. Let's start with the signature:

src/client.rs

```
75 fn handle_parameters(
76     mut builder: RequestBuilder,
77     is_form: bool,
78     is_multipart: bool,
79     parameters: &Vec<Parameter>,
80 ) -> HurlResult<RequestBuilder> {
```

We take a `RequestBuilder` and some parameter data and return a builder or an error. The `mut` in front of the `builder` argument just means to make the `builder`

argument work like a mutable local variable. We do this because the methods on `RequestBuilder` consume the receiver, i.e. they take `self` as their first argument rather than `&self` or `&mut self`, and return a new `RequestBuilder`. We could have instead continually used `let` bindings to update the `builder` variable, but we will see that using this mutable approach reduces the noise a bit.

With that out of the way, try to walk through the rest of the function. First let's setup some state to use as we loop over our parameters:

src/client.rs

```
81      let mut data: HashMap<&String, Value> = HashMap::new();
82      let mut multipart = if is_multipart {
83          Some(Form::new())
84      } else {
85          None
86      };
```

We declare a `HashMap` from strings to JSON values which will be the container for the data that we will eventually add to the request. If the request is a multipart form based on the argument passed in to us, we create a `Form` object inside an `Option` for later use. This form object is provided by reqwest for exactly this purpose of adding each part of a multipart form to a request.

Now we can loop over our parameters, and use a big a match statement to handle each one individually:

src/client.rs

```
88      for param in parameters.iter() {
89          match param {
90              Parameter::Header { key, value } => {
91                  trace!("Assing header: {}", key);
92                  builder = builder.header(key, value);
93              }
94              Parameter::Data { key, value } => {
95                  trace!("Adding data: {}", key);
96                  if multipart.is_none() {
97                      data.insert(key, Value::String(value.to_owned()));
```

```
98                  } else {
99                      multipart = multipart.map(|m| m.text(key.to_owned()\
100 , value.to_owned()));
101                 }
102             }
103             Parameter::Query { key, value } => {
104                 trace!("Adding query parameter: {}", key);
105                 builder = builder.query(&[(key, value)]);
106             }
107             Parameter::RawJsonData { key, value } => {
108                 trace!("Adding JSON data: {}", key);
109                 let v: Value = serde_json::from_str(value)?;
110                 data.insert(key, v);
111             }
112             Parameter::RawJsonDataFile { key, filename } => {
113                 trace!("Adding JSON data for key={} from file={}", key,\
114  filename);
115                 let file = File::open(filename)?;
116                 let reader = BufReader::new(file);
117                 let v: Value = serde_json::from_reader(reader)?;
118                 data.insert(key, v);
119             }
120             Parameter::DataFile { key, filename } => {
121                 trace!("Adding data from file={} for key={}", filename,\
122  key);
123                 let value = std::fs::read_to_string(filename)?;
124                 data.insert(key, Value::String(value));
125             }
126             Parameter::FormFile { key, filename } => {
127                 trace!("Adding file={} with key={}", filename, key);
128                 multipart = Some(
129                     multipart
130                         .unwrap()
131                         .file(key.to_owned(), filename.to_owned())?,
132                 );
133             }
```

```
134         }
135     }
```

The `Header` type is easy, we just call the `header` method to update our builder. The `Data` type is slightly harder as we insert the key and value into the `data` map if we are not building a multipart form, otherwise we use the methods on `Form` to add the key and value to the form. As our `multipart` variable is inside an `Option` we use `map` to operate on the `Form` type inside.

Adding query string elements is also easy given the `query` method on the builder. The `RawJsonData` type is likewise straightforward, we just have to use serde to parse the string into a `Value` before inserting into our data map.

The last three parameters all deal with files.

First, `RawJsonDataFile` parses the file into a JSON value using serde and inserts the result into our map. We use the nice facilities provided by the standard library to open a handle to a file and to build a buffered reader around that handle. We can then pass that reader directly to the `from_reader` function. This automatically handles all of the complexity of ensuring the file is available, incrementally reading it, ensuring it is closed even if something goes wrong, etc.

Second, `DataFile` reads a string from `filename` and inserts that string as a value directly in our data map. The `read_to_string` method is not what you want in many cases dealing with file I/O in Rust, but sometimes (like here), it is exactly what you need.

Lastly, we deal with `FormFile` which is quite simple thanks to the `file` function provided by the `Form` type. We call `unwrap` on `multipart` here because the existence of a `FormFile` parameter is equivalent to `is_multipart` being true so we know that in this branch of the match statement `multipart` must not be `None`.

Now that all of the parameters have been converted into more amenable data structures, we can add them to the builder:

src/client.rs

```
    if let Some(m) = multipart {
        builder = builder.multipart(m);
    } else {
        if !data.is_empty() {
            if is_form {
                builder = builder.form(&data);
            } else {
                builder = builder.json(&data);
            }
        }
    }

    Ok(builder)
}
```

If we are in the multipart form case then we use the `multipart` method, otherwise we either use the `form` or `json` methods depending on the `is_form` flag passed to us. We do the simply check to ensure that data is actually not empty before trying to serialize it as part of the request. Finally, if we have made it this far without returning an error then we know that we can successfully return our builder.

The last bit of information is dealing with authentication information. We have a helper which will also modify the builder to add the relevant data:

src/client.rs

```
fn handle_auth(
    mut builder: RequestBuilder,
    auth: &Option<String>,
    token: &Option<String>,
) -> HurlResult<RequestBuilder> {
    if let Some(auth_string) = auth {
        let (username, maybe_password) = parse_auth(&auth_string)?;
        trace!("Parsed basic authentication. Username={}", username);
        builder = builder.basic_auth(username, maybe_password);
    }
```

```
68      if let Some(bearer) = token {
69          trace!("Parsed bearer authentication. Token={}", bearer);
70          builder = builder.bearer_auth(bearer);
71      }
72      Ok(builder)
73  }
```

If we do not have an auth string or a token then we just return the builder unchanged. Otherwise, we use the `basic_auth` or `bearer_auth` methods on the builder to add the particular pieces of auth information. Our basic auth path makes use of a helper to get the username and password:

src/client.rs

```
167 fn parse_auth(s: &str) -> HurlResult<(String, Option<String>)> {
168     if let Some(idx) = s.find(':') {
169         let (username, password_with_colon) = s.split_at(idx);
170         let password = password_with_colon.trim_start_matches(':');
171         if password.is_empty() {
172             return Ok((username.to_owned(), None));
173         } else {
174             return Ok((username.to_owned(), Some(password.to_owned())));
175         }
176     } else {
177         let password = rpassword::read_password_from_tty(Some("Password\
178 : "))?;
179         return Ok((s.to_owned(), Some(password)));
180     }
181 }
```

We accept the forms `myUserName`, `myUserName:`, and `myUserName:myPassword` as valid basic auth strings. The first form without a colon assumes that you have input your username and want to type in your password. We use the extremely helpful `rpassword` crate which provides the `read_password_from_tty` method to ask the user to enter a password. This captures that value and returns it as a string without echoing the user input as you would expect.

The other two cases with colons mean that you are giving your password and we do not prompt the user to enter one. In the first case, `myUserName:`, we are saying that we explicitly want to provide no password.

The errors module

We are almost done! We have been using the errors from this module throughout the code we have written so far. It might be tempting to write your code using `unwrap` and `panic!` everywhere at first rather than propagating errors when you are just in a prototyping stage. The next best thing to that is returning results but strings for errors when you know you will need errors but you want to delay creating structured errors.

Let me caution you against both those approaches. Creating a simple error enum along with a type alias for your particular result is a tiny amount of work. Let's do that now:

src/errors.rs

```
use std::fmt;

pub enum Error {
    ParameterMissingSeparator(String),
    MissingUrlAndCommand,
    NotFormButHasFormFile,
    ClientSerialization,
    ClientTimeout,
    ClientWithStatus(reqwest::StatusCode),
    ClientOther,
    SerdeJson(serde_json::error::Category),
    IO(std::io::ErrorKind),
    UrlParseError(reqwest::UrlError),
}

pub type HurlResult<T> = Result<T, Error>;
```

We import `std::fmt` to make our `Display` implementation easier to write, but the real code here is just a simple enum and a type alias. You don't have to know about

all of these errors up front. We do here as we have a bit more foresight in a book with the rest of the code already written. In fact, most likely you will have only a couple simply variants at first. Then you will add some more which wrap errors from other libraries or the standard library. Then you will add more of your own as needed. But getting just something in place along with the type alias will make your future self much happier.

Let's beef up our error type with a couple necessities. First, we implement `Display`:

src/errors.rs

```
18 impl fmt::Display for Error {
19     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
20         match self {
21             Error::ParameterMissingSeparator(s) => {
22                 write!(f, "Missing separator when parsing parameter: {}\
23 ", s)
24             }
25             Error::MissingUrlAndCommand => write!(f, "Must specify a ur\
26 l or a command!"),
27             Error::NotFormButHasFormFile => write!(
28                 f,
29                 "Cannot have a form file 'key@filename' unless --form o\
30 ption is set"
31             ),
32             Error::ClientSerialization => write!(f, "Serializing the re\
33 quest/response failed"),
34             Error::ClientTimeout => write!(f, "Timeout during request"),
35             Error::ClientWithStatus(status) => write!(f, "Got status co\
36 de: {}", status),
37             Error::ClientOther => write!(f, "Unknown client error"),
38             Error::SerdeJson(c) => write!(f, "JSON error: {:?}", c),
39             Error::IO(k) => write!(f, "IO Error: {:?}", k),
40             Error::UrlParseError(e) => write!(f, "URL Parsing Error: {}\
41 ", e),
42         }
43     }
44 }
```

For each variant we write something meaningful, nothing really fancy here. We then implement `Debug` directly to be the same as `Display`:

src/errors.rs

```
40 impl fmt::Debug for Error {
41     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
42         write!(f, "{}", self)
43     }
44 }
```

This approach could be debated, but this mainly serves the purpose of getting the nice error message in some contexts which do a debug print by default where we do not have control.

We also want to implement the `std::error::Error` trait for our `Error` type:

src/errors.rs

```
46 impl std::error::Error for Error {
47     fn source(&self) -> Option<&(dyn std::error::Error + 'static)> {
48         match self {
49             Error::UrlParseError(e) => Some(e),
50             _ => None,
51         }
52     }
53 }
```

The custom error type world in Rust has been all over the place since Rust 1.0, however the standard library as of around 1.30 started to incorporate much of the learnings from the community so that one can often do more with just the normal tools rather than pulling another crate. However, if you are interested the following crates provide some interesting features around errors: *error-chain*, *failure*, *context-attribute*, *err-derive*, *snafu*, *fehler*, *anyhow*, and *thiserror*.

The last piece of our error puzzle is to provide implementations of the `From` trait for errors from other libraries that we want to wrap in our custom enum. There are four of these that are relevant. First, `reqwest::Error`:

src/errors.rs

```
impl From<reqwest::Error> for Error {
    #[inline]
    fn from(err: reqwest::Error) -> Error {
        if err.is_serialization() {
            return Error::ClientSerialization;
        }
        if err.is_timeout() {
            return Error::ClientTimeout;
        }
        if let Some(s) = err.status() {
            return Error::ClientWithStatus(s);
        }
        Error::ClientOther
    }
}
```

Next up is `serde_json::error::Error`:

src/errors.rs

```
impl From<serde_json::error::Error> for Error {
    #[inline]
    fn from(err: serde_json::error::Error) -> Error {
        Error::SerdeJson(err.classify())
    }
}
```

Third is `std::io::Error` for dealing with file system errors:

src/errors.rs

```
impl From<std::io::Error> for Error {
    #[inline]
    fn from(err: std::io::Error) -> Error {
        Error::IO(err.kind())
    }
}
```

Finally, we wrap `reqwest::UrlError` which comes up in URL parsing:

src/errors.rs

```
impl From<reqwest::UrlError> for Error {
    #[inline]
    fn from(err: reqwest::UrlError) -> Error {
        Error::UrlParseError(err)
    }
}
```

Recap

Phew, that was an adventure. But we can now perform all of the requests as described in the basic set of features that we intended to support. The remainder of this chapter is adding a few add-on features to make this utility slightly more useful. However, we have a surprisingly rich set of features from a relatively small codebase. This is primarily enabled by relying on the rich ecosystem of crates available to us.

Adding a configuration file

Our first extension will be to add a configuration file which we can use to set defaults across program executions. As we are writing Rust, let's use TOML for the format of our configuration file. To get started, let's bring in a few more dependencies to our `Cargo.toml` file:

Cargo.toml

```
dirs = "2.0"
lazy_static = "1.4"
toml = "0.5"
```

The `dirs` crate is a helpful utility which can give us a path to a user's home directory in a cross platform way. The `lazy_static` crate is to create static constants which are too complicated or impossible to create at compile time. Lastly, the `toml` crate is for reading/writing the TOML format.

Changes to main

We are going to make most of our changes in new modules, so let's add those at the top of our `main.rs` file:

src/main.rs

```
mod config;
mod directories;
```

Then, let's add one extra call to get the configuration data incorporated into our app by calling `process_config_file` on our app after we have parsed and validated the command line arguments:

src/main.rs

```
    let mut app = app::App::from_args();
    app.validate()?;
    app.process_config_file();
```

Adding the configuration to our app

Moving over to `app.rs`, let's bring in `PathBuf` for working with file system paths:

src/app.rs

```
3 use std::path::PathBuf;
```

Let's also bring in our `config` module so that we can reference types from it without needing the `crate` prefix everywhere:

src/app.rs

```
6 use crate::config;
```

The location of the configuration file is something we want further be able to configure so we add a `config` field to the `App` struct which optionally takes a path to look for the file. The doc comment here explains what the file is and where we will look for it:

src/app.rs

```
51     /// Configuration file.
52     ///
53     /// A TOML file which is stored by default at HOME/.config/hurl/con\
54 fig
55     /// where HOME is platform dependent.
56     ///
57     /// The file supports the following optional keys with the given ty\
58 pes:
59     /// verbose: u8
60     /// form: bool
61     /// auth: string
62     /// token: string
63     /// secure: bool
64     ///
65     /// Each option has the same meaning as the corresponding configura\
66 tion
67     /// option with the same name. The verbose setting is a number from\
68  0
69     /// meaning no logging to 5 meaning maximal log output.
```

```
70      #[structopt(short, long, env = "HURL_CONFIG", parse(from_os_str))]
71      pub config: Option<PathBuf>,
```

The new structopt attribute here is `env = "HURL_CONFIG"` which allows the user to set the location of the configuration file via the `HURL_CONFIG` environment variable in addition to the ability to pass it as a command line argument. The fact that you can get that extra bit of configurability with just a few extra characters really shows the nicety of using structopt.

The `parse(from_os_str)` attribute to get the `PathBuf` is something that is builtin to structopt as this is a very common need.

Finally, let's add the `process_config_file` method to our `App` struct:

src/app.rs

```
121     pub fn process_config_file(&mut self) {
122         let config_path = config::config_file(self);
123         let config_opt = config::read_config_file(config_path);
124         if let Some(mut config) = config_opt {
125             if self.verbose == 0 {
126                 if let Some(v) = config.verbose {
127                     self.verbose = v;
128                 }
129             }
130             if !self.form {
131                 if let Some(f) = config.form {
132                     self.form = f;
133                 }
134             }
135             if !self.secure {
136                 if let Some(s) = config.secure {
137                     self.secure = s;
138                 }
139             }
140             if self.auth.is_none() {
141                 self.auth = config.auth.take();
142             }
```

```
            if self.token.is_none() {
                self.token = config.token.take();
            }
        }
    }
```

We use helper functions from the `config` module to get the path and read the file at that path if it exists. We then use the resulting data structure, if we were able to find and parse one, to update our app struct.

The config module

The module for finding and loading the configuration file is pretty small. We first import some things we will need:

src/config.rs

```
use serde::Deserialize;
use std::fs;
use std::path::PathBuf;

use crate::app::App;
use crate::directories::DIRECTORIES;
```

Then we declare a struct to hold the data that we support in the configuration file:

src/config.rs

```
#[derive(Debug, Deserialize)]
pub struct Config {
    pub verbose: Option<u8>,
    pub form: Option<bool>,
    pub auth: Option<String>,
    pub token: Option<String>,
    pub secure: Option<bool>,
}
```

The use of `Deserialize` here is what allows the `toml` crate to be able to use the serde machineary to turn our file into an instance of this struct.

The first helper function we need is one which takes our `App` struct and returns a `PathBuf` to find a configuration file:

src/config.rs

```
17 pub fn config_file(app: &App) -> PathBuf {
18     app.config
19         .as_ref()
20         .cloned()
21         .filter(|config_path| config_path.is_file())
22         .unwrap_or_else(|| DIRECTORIES.config().join("config"))
23 }
```

If the app has a valid file defined as its config field then we use that, otherwise we use the helper provided by the directories module to give us a path to our default config directory. By using `unwrap_or_else` we ensure that we always return some `PathBuf` from this function.

Now that we have a path, we can write our helper which attempts to read and parse that file into our `Config` struct:

src/config.rs

```
25 pub fn read_config_file(path: PathBuf) -> Option<Config> {
26     fs::read_to_string(path).ok().map(|content| {
27         let config: Config = toml::from_str(&content).unwrap();
28         config
29     })
30 }
```

We turn the `Result` returned by `read_to_string` into an `Option` using `ok()` which is a common idiom when you care about failure but not the specifics of the error. The error variant just gets turned into `None` so we can use `map` on that option to be able to operate only on the case when we know we have a string of data from a file. We then use the `toml` crate along with serde to turn that string into our expected data structure.

The use of `unwrap` here is for expedience. You could choose to handle that error and return `None` in that case as well, or return a specific error instead.

The directories module

We turn now to the module responsible for doing the cross platform home directory lookup. We start with a couple familiar imports:

src/directories.rs

```
1 use lazy_static::lazy_static;
2 use std::path::{Path, PathBuf};
```

Note that the first import is bringing in the `lazy_static` macro from the `lazy_static` crate. This used to be accomplished with the `macro_use` attribute at the top level of the crate, but thankfully macros have become regular items.

We also want to include one import but only if you are building for MacOS. We can do this in Rust using the `cfg` attribute:

src/directories.rs

```
4 #[cfg(target_os = "macos")]
5 use std::env;
```

This says roughly if the target operating system is MacOS then include the subsequent item when compiling, otherwise remove it. This is a much more sane system of conditional compilation, especially if you are used to C-style `#` defines.

We create a struct to hold the default path to our configuration file:

src/directories.rs

```
7 pub struct Directories {
8     config: PathBuf,
9 }
```

If one wanted to include other directories such as to store data output then this struct could have additional members. We do it this way to make that type of modification easier.

Let's then add some methods to our `Directories` type:

src/directories.rs

```
impl Directories {
    fn new() -> Option<Directories> {
        #[cfg(target_os = "macos")]
        let config_op = env::var_os("XDG_CONFIG_HOME")
            .map(PathBuf::from)
            .filter(|p| p.is_absolute())
            .or_else(|| dirs::home_dir().map(|d| d.join(".config")));

        #[cfg(not(target_os = "macos"))]
        let config_op = dirs::config_dir();

        let config = config_op.map(|d| d.join("hurl"))?;

        Some(Directories { config })
    }

    pub fn config(&self) -> &Path {
        &self.config
    }
}
```

First, our `new` method returns a `Option` because we might not be able to find the directory we are looking for. As we discussed above, we are using the `cfg` attribute to do different things depending on if we are on MacOS or not. Here it looks like we define the variable `config_op` twice. However, depending on the target, it will only be defined once. The rest of the code can work with that value. The compiler checks that the types match in both cases so everything is safe regardless of which OS we are targeting.

The MacOS case is just trying to use a particular environment variable that is the recommended way of finding your home directory, if it is defined. Otherwise we fallback to the `dirs` crate. In other other case we just use the `dirs` crate directly.

After that we add `hurl` to the end of the path and stick that path inside the `Directories` struct that we return.

We also create a convenience method which turns the `PathBuf` into a `Path` by the magic of the `Deref` trait.

Finally we use `lazy_static` to expose a static reference to a newly constructed `Directories` struct:

src/directories.rs

```
32 lazy_static! {
33     pub static ref DIRECTORIES: Directories =
34         Directories::new().expect("Could not get home directory");
35 }
```

We use `expect` here to crash if we cannot get a path to the home directory. This only occurs we literally cannot construct a path, it is not about whether the config directory exists or whether the config file exists. Without a home directory there is not much we can do when it comes to configuration files. This is a further opportunity to clean up as this case could be more gracefully handled and probably just ignored.

Adding sessions

A common pattern that comes up when testing APIs on the command line is to make multiple requests that depend on one another. The most common dependency comes from authentication where we login as a user and then want to perform some actions on their behalf. A session is the common terminology for a set of activity between logging in and logging out. Therefore we will use this name for the concept of sharing state across requests.

As we said the primary use case for sessions is authentication but we might also want to always set a specific header. This is distinct from our configuration file as it is more specific to a particular group of requests and can also be modified by the response.

Like the configuration change, we will thread the session through the app starting in main and ending in a session module where most of the details are implemented.

Adding the session to main

The easy bit is adding are declaration that there will be a session module:

src/main.rs

```
8  mod directories;
9  mod errors;
10 mod session;
```

Then, inside our main function, after setting up logging, but before we match on the `cmd` argument, we will try to get a session:

src/main.rs

```
21     if let Some(level) = app.log_level() {
22         std::env::set_var("RUST_LOG", format!("hurl={}", level));
23         pretty_env_logger::init();
24     }
25
26     let mut session = app
27         .session
28         .as_ref()
29         .map(|name| session::Session::get_or_create(&app, name.clone(),\
30  app.host()));
31
32     match app.cmd {
```

It may not be clear from this code due to type inference but the session variable here has type `Option<Session>`. We will have some type of `session` field on the app which is optional and if it exists then we will turn that into a `Session`.

We then move to `handle_response`:

src/main.rs

```
fn handle_response(
    app: &app::App,
    mut resp: reqwest::Response,
    session: &mut Option<session::Session>,
) -> HurlResult<()> {
```

We add the `app` and `session` to this function as the response might update the session. Inside this function let's take care of that update:

src/main.rs

```
    if !app.read_only {
        if let Some(s) = session {
            s.update_with_response(&resp);
            s.save(app)?;
        }
    }
    Ok(())
```

This is just after we have finished all of our output and are about to return. If the session is not read only and the session exists, then we update the session from the response and save the session.

We need to update the calls to `handle_response` and in the process we also want to update our calls to `perform_method` and `perform` to also take the session. First, let's update the case where `cmd` is exists:

src/main.rs

```
        Some(ref method) => {
            let resp = client::perform_method(&app, method, &mut sessio\
n)?;
            handle_response(&app, resp, &mut session)
        }
```

Then we can update the situation where `cmd` is None:

src/main.rs

```
        None => {
            let url = app.url.take().unwrap();
            let has_data = app.parameters.iter().any(|p| p.is_data());
            let method = if has_data {
                reqwest::Method::POST
            } else {
                reqwest::Method::GET
            };
            let resp = client::perform(&app, method, &mut session, &url\
, &app.parameters)?;
            handle_response(&app, resp, &mut session)
        }
```

We pass a mutable reference to the perform functions because we need the session to possibly fill in data in the request and we also will fill in the session with other data provided as part of the request as parameters. This will be more clear when we get to the client module, but for now this is just plumbing.

Supporting sessions in the app module

The app module is responsible for command line argument based configuration. Therefore our primary responsibility here is to add the fields to the `App` struct necessary for configuring the session. Let's add those fields:

src/app.rs

```
44      /// Session name.
45      #[structopt(long)]
46      pub session: Option<String>,
47
48      /// Session storage location.
49      #[structopt(long, parse(from_os_str))]
50      pub session_dir: Option<PathBuf>,
51
52      /// If true then use the stored session to augment the request,
53      /// but do not modify what is stored.
54      #[structopt(long)]
55      pub read_only: bool,
```

The `session` field is a name to use for a particular session. Specifying the same name is how sessions get reused across requests. We also provide the ability to specify where the session data should be stored. By default we will put things in our configuration directory for convenience, but this gives the user an option to put it somewhere else. Finally, the `read_only` field determines whether the session should be modified by the request and response or if it should only be used to augment the request as it currently exists on disk.

While we are here, let's add one helper method to the app struct which we will use when we try to store the session. First we bring in one useful helper from the session module:

src/app.rs

```
8   use crate::session::make_safe_pathname;
```

Then we add the `host` method to the app:

src/app.rs

```
177     pub fn host(&self) -> String {
178         if let Some(url) = &self.url {
179             make_safe_pathname(url)
180         } else if let Some(cmd) = &self.cmd {
181             make_safe_pathname(&cmd.data().url)
182         } else {
183             unreachable!();
184         }
185     }
```

This is a simple way to get some string representation of the URL we are making requests to. Sessions are unique based on this host value and the configured named.

Adding sessions to the client

Recall that the client is responsible for actually making the network request. The session will be part of building up the request and as we saw is part of our perform methods. Therefore, let's start with importing the session type:

src/client.rs

```
4 use crate::session::Session;
```

Then we can add the session to `perform_method` as we saw before at the call site in `main.rs`:

src/client.rs

```
pub fn perform_method(
    app: &App,
    method: &Method,
    session: &mut Option<Session>,
) -> HurlResult<Response> {
```

As this just delegates to `perform`, we can just pass the passed in session along directly:

src/client.rs

```
    perform(
        app,
        method.into(),
        session,
        &method_data.url,
        &method_data.parameters,
    )
```

Our `perform` method needs the signature updated to take the session:

src/client.rs

```
pub fn perform(
    app: &App,
    method: reqwest::Method,
    session: &mut Option<Session>,
    raw_url: &str,
    parameters: &Vec<Parameter>,
) -> HurlResult<Response> {
```

We then need to modify our request builder from the session, and possible modify the session based on the other data available. We do this by creating a function called `handle_session` which takes the builder and all of the pertinent data and returns a new builder. We call this right after creating our builder:

src/client.rs

```
    let mut builder = client.request(method, url);
    builder = handle_session(
        builder,
        session,
        parameters,
        !app.read_only,
        &app.auth,
        &app.token,
    );
```

Let's implement `handle_session` mostly by delegating to methods on the session:

src/client.rs

```
fn handle_session(
    mut builder: RequestBuilder,
    session: &mut Option<Session>,
    parameters: &Vec<Parameter>,
    update_session: bool,
    auth: &Option<String>,
    token: &Option<String>,
) -> RequestBuilder {
    if let Some(s) = session {
        trace!("Adding session data to request");
        builder = s.add_to_request(builder);
        if update_session {
            trace!("Updating session with parameters from this request"\
);
            s.update_with_parameters(parameters);
            s.update_auth(auth, token);
        }
    }
    builder
}
```

If we have a session, then we update the builder by adding the session data to the request. Furthermore, if we are not in read only mode which means we want to update the session, then we pass the parameters and authentication information to the session for updating.

Implementing sessions

Finally we get to the meat of building sessions. We have many small tasks to accomplish in the session module which when put together will do everything we need. Let's start with a big stack of imports:

src/session.rs

```
use crate::app::{App, Parameter};
use crate::directories::DIRECTORIES;
use crate::errors::HurlResult;
use reqwest::header::COOKIE;
use reqwest::RequestBuilder;
use serde::{Deserialize, Serialize};
use std::collections::HashMap;
use std::fs::{create_dir_all, File, OpenOptions};
use std::io::{BufReader, BufWriter};
use std::path::PathBuf;
```

We will get to each of these as they come up, but let's move on to defining the struct to contain the session data:

src/session.rs

```
#[derive(Debug, Default, Serialize, Deserialize)]
pub struct Session {
    path: PathBuf,
    name: String,
    host: String,
    auth: Option<String>,
    token: Option<String>,
    headers: HashMap<String, String>,
```

```
20      cookies: Vec<(String, String)>,
21  }
```

We derive `Serialize` and `Deserialize` because we are going to read and write this struct to a file. The `path`, `name`, and `host` are used uniquely describe the file system location for the session. The other fields: `auth`, `token`, `headers`, and `cookies` should be self-explanatory.

Most of our functionality will live as methods implemented on this struct, so let's get started with `new`:

src/session.rs

```
23  impl Session {
24      pub fn new(app: &App, name: String, host: String) -> Self {
25          let path = Session::path(app, &name, &host);
26          Session {
27              path,
28              name,
29              host,
30              ..Default::default()
31          }
32      }
```

Constructing a new session is accomplished by using a helper method to get a path and then storing the data we are given inside a new session struct and using default values for the rest. The syntax `..Default::default()` inside a struct literal is known as the struct update syntax. It means to use the values from a default constructed instance of `Session` to fill in any fields which are not provided above.

The `Default::default()` bit is not special, but as we derive `Default` on our struct it is just one way to create a full `Session` struct with default values. If `other_session` was a variable with type `Session`, then you could write `..other_session` to use the values from `other_session` to fill in the missing values. You can think of it as updating the struct named after the `..` with the values explicitly provided.

Before we get to `path`, let's finish our construction methods by moving on to `load` which loads a `Session` from a file:

src/session.rs

```
34      pub fn load(app: &App, name: &str, host: &str) -> HurlResult<Self> {
35          let path = Session::path(app, name, host);
36          let file = File::open(path)?;
37          let reader = BufReader::new(file);
38          serde_json::from_reader(reader).map_err(|e| e.into())
39      }
```

We use `path` again to get the path, then we do the standard sequence of opening a file, putting a reader around the file handle, and then using serde to deserialize a type from the data in the file. Here serde knows what to turn the data in the file into because of the return type of the function.

Finally, we can implement `get_or_create` which is the convenient method we used in `main` which puts together `load` and `new`:

src/session.rs

```
40      pub fn get_or_create(app: &App, name: String, host: String) -> Self\
41  {
42          match Session::load(app, &name, &host) {
43              Ok(session) => session,
44              Err(_) => Session::new(app, name, host),
45          }
46      }
```

Now let's see how we get a consistent path for storing the session file:

src/session.rs

```
48      fn path(app: &App, name: &str, host: &str) -> PathBuf {
49          let mut session_dir = Session::dir(app, host);
50          let mut filename = make_safe_pathname(name);
51          filename.push_str(".json");
52          session_dir.push(filename);
53          session_dir
54      }
```

We use another helper to get the directory which we combine with the result of `make_safe_pathname` applied to the name of the session to get a full path to the file where we want to store the session.

The `dir` helper starts with getting a directory based on the app configuration or using a default, and then adds a path based on the host:

src/session.rs

```
56      fn dir(app: &App, host: &str) -> PathBuf {
57          let mut session_dir = app
58              .session_dir
59              .as_ref()
60              .cloned()
61              .filter(|session_dir| session_dir.is_dir())
62              .unwrap_or_else(|| DIRECTORIES.config().join("sessions"));
63          session_dir.push(make_safe_pathname(host));
64          session_dir
65      }
```

This is how we combine the host and session name to get consistent session file.

The session needs to be able to save itself to disk, so we implement `save` on an instance of session:

src/session.rs

```
67      pub fn save(&self, app: &App) -> HurlResult<()> {
68          let dir = Session::dir(app, &self.host);
69          create_dir_all(dir)?;
70          let file = OpenOptions::new()
71              .create(true)
72              .write(true)
73              .truncate(true)
74              .open(&self.path)?;
75          let writer = BufWriter::new(file);
76          serde_json::to_writer(writer, &self).map_err(|e| e.into())
77      }
```

We start by ensuring the directory where we want to store the file exists. The function `create_all_dir` is effectively `mkdir -p` which creates all intermediate directories as needed. We then open the file at our specified path and serialize our data structure to disk as JSON. We use the `OpenOptions` builder to specify exactly how we want the file opened. We want the file on disk to exactly represent the current struct we are trying to write in both cases where the file does and does not exist. We need `create` for the does not exist case, and we need `truncate` to properly handle all cases when the file does exist.

That takes care of all of the functionality of reading/writing the session to disk. We turn now to how we actually use the session to modify the request and how we modify the session from the request and response.

First let's deal with updating the session from the request parameters:

src/session.rs

```
79      pub fn update_with_parameters(&mut self, parameters: &Vec<Parameter\
80  >) {
81          for parameter in parameters.iter() {
82              match parameter {
83                  Parameter::Header { key, value } => {
84                      let lower_key = key.to_ascii_lowercase();
85                      if lower_key.starts_with("content-") || lower_key.s\
86  tarts_with("if-") {
87                          continue;
88                      }
89                      self.headers.insert(key.clone(), value.clone());
90                  }
91                  _ => {}
92              }
93          }
94      }
```

We have chosen to only support headers as a parameter that we store in the session so that is the only parameter type that we care about. We store the headers in the `headers` field on the session, but we exclude a few keys as persisting them across request is almost certainly not what you want to do.

Next we update the session with auth data if it is available:

src/session.rs

```
94      pub fn update_auth(&mut self, auth: &Option<String>, token: &Option\
95  <String>) {
96          if auth.is_some() {
97              self.auth = auth.clone();
98          }
99
100         if token.is_some() {
101             self.token = token.clone();
102         }
103     }
```

We have two remaining tasks. One is to add the session to the request. The other is to update the session from the response. Let's first update the request with the session:

src/session.rs

```
104     pub fn add_to_request(&self, mut builder: RequestBuilder) -> Reques\
105 tBuilder {
106         for (key, value) in self.headers.iter() {
107             builder = builder.header(key, value);
108         }
109         let cookies = self
110             .cookies
111             .iter()
112             .map(|(name, value)| format!("{}={}", name, value))
113             .collect::<Vec<String>>()
114             .join("; ");
115         if cookies.is_empty() {
116             return builder;
117         }
118         builder.header(COOKIE, cookies)
119     }
```

IF we have headers, we add them to the request. Further, if we have cookies, we turn them into the expected format for the cookie header and add that to the request. The format of the cookie header is given by the HTTP specification and the key for the cookie header is provided by reqwest as the constant `COOKIE`.

The only part of the response that we want to absorb into the session is the cookies, so let's write the function for updating the session accordingly:

src/session.rs

```
120     pub fn update_with_response(&mut self, resp: &reqwest::Response) {
121         for cookie in resp.cookies() {
122             self.cookies
123                 .push((cookie.name().to_owned(), cookie.value().to_owne\
124 d()));
125         }
126     }
127 }
```

Lastly, we have used the helper `make_safe_pathname` in a few places, but we still have to write it:

src/session.rs

```
128 pub fn make_safe_pathname(s: &str) -> String {
129     let mut buf = String::with_capacity(s.len());
130     for c in s.chars() {
131         match c {
132             'a'..='z' | 'A'..='Z' | '0'..='9' | '_' | '-' | ' ' => buf.\
133 push(c),
134             _ => buf.push('_'),
135         }
136     }
137     buf
138 }
```

This is one choice for turning a string into something that is safe for storing on the file system. You can choose your own scheme if you want, but this demonstrates one approach that works.

Session recap

We can now execute the following requests

```
$ hurl --session foo POST example.com X-API-TOKEN:abc123 info=fuzz
$ hurl --session foo example.com/info
```

and the second request will include the `X-API_TOKEN:abc123` header. This is a really nice feature that you either realize you've always wanted or will some day.

Syntax highlighting

One final feature to add a bit of polish to our little application: syntax highlighting. We are going to implement this in a way that is not completely general to save on space, but all the ideas are still here. Moreover, as most syntax highlighting is designed around programming languages, we will have to do some custom work to support highlighting our output.

Let's start with adding one more dependency to our manifest:

Cargo.toml

```
syntect = "3.2"
```

Adding highlighting to main

We start by importing some types related to highlighting and declare a `syntax` module which will encapsulate our custom syntax:

src/main.rs

```
use syntect::highlighting::Theme;
use syntect::parsing::SyntaxSet;

mod syntax;
```

Early on in our main function, we are going to build a syntax set and a theme:

src/main.rs

```
26      if let Some(level) = app.log_level() {
27          std::env::set_var("RUST_LOG", format!("hurl={}", level));
28          pretty_env_logger::init();
29      }
30
31      let (ss, ts) = syntax::build()?;
32      let theme = &ts.themes["Solarized (dark)"];
33
34      let mut session = app
35          .session
```

A `SyntaxSet` is exactly what it sounds like, it is a set of grammatical syntaxes which can be highlighted. You can think about it as a way to go from raw text to structured text. A theme is a description of what colors to apply based on the annotations in that structured text.

Our syntax module will expose a build function that takes care of this, but it exposes both a set of syntaxes as well as a set of themes. To get one theme we have to ask for one particular theme from the theme set. Here we cheat a bit and hard code one that is included by default by the `syntect` crate. This is one opportunity for an extension by making this configurable.

The response is what we want to highlight, so we add our syntax set and theme to `handle_response`:

src/main.rs

```
58  fn handle_response(
59      app: &app::App,
60      ss: &SyntaxSet,
61      theme: &Theme,
62      mut resp: reqwest::Response,
63      session: &mut Option<session::Session>,
64  ) -> HurlResult<()> {
```

Instead of calling `println!`, we are going to call `highlight_string` which will still print the string but with colors:

src/main.rs

```
88      s.push_str(&(&headers[..]).join("\n"));
89      highlight_string(ss, theme, "HTTP", &s);
```

We need to add a call to `println!("")` to add a newline between the header content and the body:

src/main.rs

```
91      println!("");
92      let result_json: serde_json::Result<OrderedJson> = serde_json::from\
93  _str(&result);
```

If we managed to parse the body as JSON then we replace to print call again with `highlight_string`:

src/main.rs

```
95              let result_str = serde_json::to_string_pretty(&result_value\
96  )?;
97              highlight_string(ss, theme, "JSON", &result_str);
```

We need to pass the syntax set and theme to `handle_response` in the case we have a `cmd`:

src/main.rs

```
41              let resp = client::perform_method(&app, method, &mut sessio\
42  n)?;
43              handle_response(&app, &ss, theme, resp, &mut session)
```

and in the case where we do not have a `cmd`:

src/main.rs

```
            let resp = client::perform(&app, method, &mut session, &url\
, &app.parameters)?;
            handle_response(&app, &ss, theme, resp, &mut session)
```

Finally, we can implement the real highlighting:

src/main.rs

```
fn highlight_string(ss: &SyntaxSet, theme: &Theme, syntax: &str, string\
: &str) {
    use syntect::easy::HighlightLines;
    use syntect::util::{as_24_bit_terminal_escaped, LinesWithEndings};

    let syn = ss
        .find_syntax_by_name(syntax)
        .expect(&format!("{} syntax should exist", syntax));
    let mut h = HighlightLines::new(syn, theme);
    for line in LinesWithEndings::from(string) {
        let regions = h.highlight(line, &ss);
        print!("{}", as_24_bit_terminal_escaped(&regions[..], false));
    }
    println!("\x1b[0m");
}
```

The `syntect` crate has a really nice set of features because it provides an incredible amount of intricate customizations and details for syntax highlighting, but it also provides some easy convenience methods when you don't need to get too deep into the details.

We name our two syntaxes `"JSON"` and `"HTTP"` and pass those as the `syntax` argument to this function so that we can easily look up the right syntax from our syntax set.

We use this syntax and theme to construct a `HighlightLines` object which is used to generate text in colored regions for printing. The `as_24_bit_terminal_escaped` method is a nice utility for turning those colored regions into escape sequences for use in terminal output. The `LinesWithEndings` will add newlines if they don't exist so

that we know that we can use `print!` instead of `println!` and still will get newlines at the end of our lines.

Finally, we print out the terminal reset character which ends all highlighting and puts us back in to normal shell mode so that we don't leak our highlighting onto later shell commands.

Supporting syntax highlighting errors

We want to gracefully handle failing to load syntax definitions, so let's add a variant to our error enum to represent this failure:

src/errors.rs

```
12      IO(std::io::ErrorKind),
13      UrlParseError(reqwest::UrlError),
14      SyntaxLoadError(&'static str),
15  }
```

The only other thing we need to do is add support for printing this error:

src/errors.rs

```
36              Error::UrlParseError(e) => write!(f, "URL Parsing Error: {}\
37  ", e),
38              Error::SyntaxLoadError(typ) => write!(f, "Error loading syn\
39  tax for {}", typ),
```

Luckily, if you somehow forgot to do that the compiler would give you an error because the match statement in the `Debug` implementation would no longer be exhaustive.

The syntax module

We have one purpose in this module, to load syntax and theme sets. Let's add the relevant imports to this new `src/syntax.rs` file:

src/syntax.rs

```
use crate::errors::{Error, HurlResult};
use syntect::highlighting::ThemeSet;
use syntect::parsing::syntax_definition::SyntaxDefinition;
use syntect::parsing::{SyntaxSet, SyntaxSetBuilder};
```

We expose one public function, `build`, which returns a syntax set and a theme set:

src/syntax.rs

```
pub fn build() -> HurlResult<(SyntaxSet, ThemeSet)> {
    let mut builder = SyntaxSetBuilder::new();
    let http_syntax_def = include_str!("../HTTP.sublime-syntax");
    let def = SyntaxDefinition::load_from_str(http_syntax_def, true, No\
ne)
        .map_err(|_| Error::SyntaxLoadError("HTTP"))?;
    builder.add(def);

    let json_syntax_def = include_str!("../JSON.sublime-syntax");
    let json_def = SyntaxDefinition::load_from_str(json_syntax_def, tru\
e, None)
        .map_err(|_| Error::SyntaxLoadError("JSON"))?;
    builder.add(json_def);
    let ss = builder.build();

    let ts = ThemeSet::load_defaults();
    Ok((ss, ts))
}
```

Syntect uses a builder pattern for constructing a syntax set. We first create the HTTP syntax and then the JSON syntax. For the theme we just use the provided `load_defaults` method to get a decent set of themes to return.

The code to construct each syntax definition is the same except for the path to the file which is brought in via `include_str!`. The path used here is relative to the file in which this macro exists. As we are inside the `src` directory, this implies

that `../HTTP.sublime-syntax` is therefore located in the same directory as our `Cargo.toml` file.

The `include_str!` macro is extremely handy. This reads a file at compile time and includes the bytes directly in the binary as a `&'static str`. This allows you to have the benefit of keeping data separate in a file, without the cost of having to read it at runtime. Obviously this makes your binary bigger, but this is such a great tool when you want to make that trade-off.

The two syntax definition files are in the sublime syntax format which is what Syntect uses. This format is a YAML file which describes how to assign predefined highlight attributes to pieces of text based on regular expressions. The HTTP syntax we define does some highlighting of the version, status, and headers:

HTTP.sublime-syntax

```
%YAML 1.2
---
name: HTTP
file_extensions:
  - http
scope: source.http
contexts:
  main:
    - match: ([A-Z]+)( +)([^ ]+)( +)(HTTP)(/)(\d+\.\d+)
      captures:
        1: function.name.http
        2: string.quoted.double.http
        3: entity.name.section.http
        4: string.quoted.double.http
        5: keyword.http
        6: keyword.operator.http
        7: constant.numeric.integer.decimal.http
    - match: (HTTP)(/)(\d+\.\d+)( +)(\d{3})( +)(.+)
      captures:
        1: keyword.http
        2: keyword.operator.http
        3: constant.language.http
```

```
        4: string.quoted.double.http
        5: constant.numeric.integer.decimal.http
        6: string.quoted.double.http
        7: keyword.symbol.http
    - match: (.*?)( *)(:)( *)(.+)
      captures:
        1: function.name.http
        2: string.quoted.double.http
        3: keyword.operator.http
        4: string.quoted.double.http
        5: string.quoted.double.http
```

This syntax is completely custom for this application and could easily be extended or changed based on what you might rather see.

The JSON syntax is similar but much longer. You can find various packages as part of the Sublime editor repo. For example, a good JSON syntax can be found here:

> https://github.com/sublimehq/Packages/blob/v3186/JavaScript/JSON.sublime-syntax

Summary

We built a fraction of the features of cURL but we added some nice things on top like syntax highlighting and sessions. Although there is quite a bit of code that we had to write, we still only wrote a tiny fraction compared to the amount of work that is done for us by the great libraries that we built on top of.

The ecosystem of crates has matured to the point that you can probably find a solid library to accomplish most of the mundane tasks. The fun part is putting them all together to build interesting, useful, and new experiences.

Macros

Overview

Writing code that generates code is known as metaprogramming. Some languages have no native support for this, while others accomplish it textual substitution, and still others have the ability to operate on syntax as data. This last concept comes from Lisp and is called a macro. In short, a macro is a thing which takes code as input and produces code as output. Rust has a set of macro features depending on what you are trying to accomplish and how much work you want to undertake. These are:

- declarative macros
- procedural custom derive macros
- procedural attribute macros
- procedural function-like macros

Macros serve to reduce the amount of code you need to write, but it is important to understand that anything you can do with a macro you could have done by hand. There is nothing magic going on in the world of macros even if it sometimes feels otherwise. That is not to understate the enormous power of metaprogramming. Generating code can be a significant multiplier to your productivity and is absolutely the right tool for some jobs.

But, as always, with great power comes great responsibility. Code that generates code can be hard to understand, hard to change, and hard to debug. Frequently macros are not the right tool to reach for. If you can get away with using a function or some other abstraction mechanism for reducing repetition, then you should almost certainly use that instead of a macro.

There are two primary reasons to reach for a macro. The first is if you want a function with a variable number of arguments. The second reason is if you want to do something that must happen at compile time, like implement a trait. It is impossible

to write a normal function that does either of these in regular Rust, but both can be accomplished with macros.

We are going to cover declarative macros in a limited fashion as they are the less powerful but better documented type of macros. If you can accomplish your task easily with a declarative macro, then fantastic. However, they can get hairy very quickly. Moreover, as we will see, the more powerful procedural macros are actually closer to writing regular Rust and are therefore more approachable for bigger problems.

Declarative Macros

The name of this type of macro comes from the fact that the language used to implement these is a declarative style language inspired by Scheme. You can think of it as being similar to a big `match` statement where the conditions are matching on syntactic constructs and the result is the code you want to generate.

Rather than explain all the syntax and go through a bunch of boilerplate, let's first just write a macro that is similar to something you have seen before. Suppose the standard library did not come with the `vec!` macro. We find ourselves writing code like:

```
let mut v = Vec::new();
v.push(1);
v.push(2);
v.push(3);
let v = v;
```

This starts to get tedious and it requires that weird `let v = v;` line to get an immutable binding to our vector after we are done filling it up. We could have instead put the creation code in a block such as:

```
let v = {
  let mut a = Vec::new();
  a.push(1);
  a.push(2);
  a.push(3);
  a
};
```

In some ways that is a little better, but we still have a lot of repetition going on. Can we instead write some function that does this for us? That would look like:

```
let v = make_vector(1, 2, 3);
```

This implies this signature:

```
fn make_vector(x0: i32, x1: i32, x2: i32) -> Vec<i32>
```

That would work if we only wanted to create three elements vectors of type `i32`. It is easy to make the type more general by using a generic:

```
fn make_vector<T>(x0: T, x1: T, x2: T) -> Vec<T>
```

But can we extend this to an arbitrary number of elements? Rust does not support functions with a variable number of arguments, commonly called variadic functions. There is no way to write a function in Rust that takes an unknown number of arguments. We could write out many similar functions:

```
fn make_vector_1<T>(x0: T) -> Vec<T>
fn make_vector_2<T>(x0: T, x1: T) -> Vec<T>
fn make_vector_3<T>(x0: T, x1: T, x2: T) -> Vec<T>
...
```

Depending on your use case, like if you are constantly making only one size vector but you do it in a lot of places, then maybe that could work. But this certainly does not scale. Enter macros which execute at compile time and are therefore not bound by the same constraints as the rest of the Rust language.

Let's implement this as a macro called `myvec`:

main.rs

```
macro_rules! myvec {
    ($($x:expr),*) => ({
        let mut v = Vec::new();
        $(v.push($x);)*
        v
    });
    ($($x:expr,)*) => (myvec![$($x),*])
}
```

The syntax for creating a macro is to invoke a macro called `macro_rules` (very meta) with the name of your new macro, `myvec` in this case, without the exclamation point, followed by braces to denote the body of the macro.

The body of a macro is just a series of rules with the left side of an arrow, `=>`, indicating what pattern to match, and the right side specifying what code to generate. The patterns are checked in order from top to bottom, so they move from more specific to more general so that the specific patterns get a chance to match.

We have two possible match arms, let's start by explaining the first one. `($($x:expr),*)` The outer set of parentheses exists to denote the entire pattern. Inside we see `$($x:expr),*` which can be broken down into two parts. The inside `$x:expr` means to match a Rust expression and bind it to the variable `$x`. The outer part `$(...),*` means to match zero or more comma separated things inside the parentheses. So the total pattern means to match zero or more comma separated expressions, with each expression bound to the variable `$x`. We will see how that is possible to bind multiple expressions to a single name when we get to the right hand side.

The right hand side is surrounded by parentheses as well which signifies encloses the entirety of the code to generate. We then also have curly braces surrounding our code, these mean to literally generate a set of curly braces around our code so that we are actually outputting a block. If you did not have these then the macro would expand to just the literal lines of code. Sometimes that is what you want, but for us as we are going to use this macro as the right hand side of an assignment, we want it to expand to a single expression.

The macro definition grammar is a bit wild. As our first example of this, the outer parentheses can actually be any balanced bracket, i.e. `()`, `[]`, or `{}` are all acceptable

in that position. So sometimes you might see `($($x:expr),*) => {{ ... }}` which still means to generate a single block, as the outer braces is just there so that the compiler knows where the right hand side begins and ends.

Then the rest of the right hand side looks just like the example code that we are replacing, except for the funky bit in middle. We are creating a vector, doing what looks kinda like our push, and then returning that vector. On the right hand side of a match inside a macro definition the syntax `$(...)*` means to repeat the code inside the parentheses for each repetition captured in the match arm. Within those parentheses the expression that we captured will be substituted directly for `$x`. It is in this way of expanding a repetition on the right that we get access to each of the repeated captures on the left.

The code inside that repetition `v.push($x);` is exactly what we were using before if you mentally replace `$x` with the different expressions that we pass to our macro.

From just what we have covered so far, we can understand that:

```
let a = myvec![1, 2, 3];
```

will expand to

```
let a = {
  let mut v = Vec::new();
  v.push(1);
  v.push(2);
  v.push(3);
  v
};
```

which is exactly the code we saw earlier in our motivation to write this macro. What about if we wrote:

```
let a = myvec![1, 2, 3,];
```

Note the trailing comma after the `3`. This is where the second match arm comes in. It turns out that the syntax in the first match arm `($(...),*)` implies matching

a repetition of what is inside `$(...)` exactly separated by commas, i.e. without a trailing comma. Without the second part of our macro, having a trailing comma would be a syntax error as the first arm does not match and it is an error to have a macro without any arms that match.

Our second pattern `$($x:expr,)*` with the comma inside the repetition, `$(...)*`, means that we expect to see expressions followed by commas. This arm therefore only matches if we explicitly do have a trailing comma.

We use the right hand side to convert the trailing comma version to the version without a trailing comma and then rely on our previous pattern to match. We do this by recursively calling our macro and expanding `$($x:expr,)*` to `$($x),*`. Moving the comma from inside the repetition to outside means to take it from a list of expressions each followed by a comma to a list of expressions separated by commas.

With this little syntactic trampoline we get a macro which supports optional trailing commas. Rust generally supports trailing commas so your macros should too. It is a bit unfortunate that you need to go through this hoop for all macros to support both styles. Further, it is reasonable to feel like this syntax is weird and really not like the rest of Rust. You are correct. This declarative style of macro will probably be deprecated at some point in a future Rust version.

Expanding a macro

Sometimes you think your macro is not quite working how you expect but you can't quite tell why. It would be great if you could see what the macro expands into to see if the generated code is what you expect. Look no further than the cargo expand[49] command. There are installation instructions on the Github repo for the project, but the short story is `cargo install cargo-expand` should allow you to then run `cargo expand` from the root of a crate to see all macros expanded out. This works for all types of macros including procedural macros which we will cover later.

You must have a nightly toolchain installed for the expansion to work. You just need it installed, you don't have to be using it directly, the command will find it as needed as long as you have it installed. This can be accomplished using `rustup`.

Consider the following code in our `main.rs` that includes the `myvec` macro definition:

[49] https://github.com/dtolnay/cargo-expand

main.rs

```
fn main() {
    let a = myvec![1, 2, 3, 4,];

    let aa = vec![1, 2, 3, 4,];
}
```

We can run `cargo expand` from the root of this crate to see:

```
#![feature(prelude_import)]
#[prelude_import]
use std::prelude::v1::*;
#[macro_use]
extern crate std;
fn main() {
    let a = {
        let mut v = Vec::new();
        v.push(1);
        v.push(2);
        v.push(3);
        v.push(4);
        v
    };
    let aa = <[_]>::into_vec(box [1, 2, 3, 4]);
}
```

We see our macro expands out to exactly what we expect based on walking through the code. For comparison we also include a call to the `vec!` macro which is in the standard library. This expands to calling an inherent method on the slice type that gets passed syntax for a boxed slice literal slice.

If you keep digging through the source you can eventually work out the entire implementation at play here. It only uses normal functions so there are no more macros hidden by this function call. This approach is technically more performant than what we are doing at the expense of a little clarity.

More information

The Rust reference[50] has a section on "macros by example" which covers the grammar of this type of macro in more detail. In particular, besides expressions, you can match most of the other parts of Rust syntax like statements and types. There is also a good explanation for matching multiple related repeated items and how you can sort out that repetition.

For a more complex example, see this blog post[51] which provides an example of computing expressions in reverse polish notation at compile time using a declarative macro.

Procedural Macros

Declarative macros were the original type of macro that one could write in stable Rust, but for a long time procedural macros were used internally in the compiler as well as in an unstable form with the nightly toolchain. As of Rust 1.30 this type of macro became available on stable.

There are three types of procedural macros:

- Custom derive
- Attribute-like
- Function-like

It is easier to see examples of these than to try to explain them theoretically.

Custom derive

First, custom derive means that you can create define a trait `MyCoolTrait` and write a macro that allows the following code to work:

[50] https://doc.rust-lang.org/1.30.0/reference/macros-by-example.html
[51] https://blog.cloudflare.com/writing-complex-macros-in-rust-reverse-polish-notation/

```
#[derive(MyCoolTrait)]
struct SomeStruct;
```

These means that `SomeStruct` will implement `MyCoolTrait` automatically by having the implementation generated at compile time. This works by sending the syntax of the item the derive is placed on to some code that you write which in turn returns new syntax which will be added to the source alongside the item. We will see an example of this when we write our own custom derive later.

Attribute-like

Attributes are the annotations on items inside the syntax `#[...]`. For example, `#[derive(Debug)]` is an attribute, it is the derive attribute which takes arguments. Unlike the custom derive macros which allow us to define how arguments to the derive attribute operator, we can instead create new attributes. One example we saw when working on building a web server was defining route information:

```
#[get("/lookup/{index}")]
fn lookup(...) {
```

The `get` attribute is custom and is implemented via a procedural macro. This type of macro is a function that takes the arguments to the attribute as raw syntax as well as the item it is being defined on as syntax and then generates code. Attributes can go in a lot of different places, a lot more than where the derive attribute is valid, so these are a pretty powerful mechanism for extending the Rust syntax.

Function-like

This type of procedural macro is maybe less obvious as you might expect something like the following:

```
let sql = sql!(SELECT * FROM users WHERE id=4);
```

This is a form of function-like procedural macro, however function-like macros are not currently stable in expression or statement position. It is expected that this will stabilize at some not too distant point in the future, but as of Rust 1.40 we still do not have this feature.

However, using the proc-macro-hack[52] crate you can get function-like procedural macros in expression and statement positions. This is a hack because it requires creating a couple extra crates to get everything working.

The type of function-like procedural macro that you can do on stable today looks like:

```
gen_object! {
  class Foo: SomeThing {
    x: u32,
    y: RefCell<i16>,
  }

  impl Foo {
    ...
  }
}
```

This means that `gen_object` takes all of the subsequent syntax as input and then generates new code to replace it.

We will concentrate on one type of procedural macro although the others use similar techniques so doing one should inform you quite a bit about all types of procedural macros.

Writing a custom derive

Our goal for the remainder of this chapter is to implement a procedural macro that can be used as an argument to the derive attribute. The most important step in writing a macro is to know that you need one.

[52] https://github.com/dtolnay/proc-macro-hack

Primarily this happens when you find yourself writing repetitive, tedious code. You almost certainly need to write that code first however. If you jump straight to a macro, you are going to have a bad time when you try to figure out what exactly you want to generate. This is similar to our `myvec!` declarative macro above, where we first wrote out the code by hand for a particular example that we wanted to replace.

Motivation

The builder pattern is one of the classic "Gang of Four" design patterns that is typically thought of in the context of object-oriented code. However, it has a place elsewhere, and can be quite useful in some problem domains. The pattern separates construction from the actual representation of an object.

So instead of writing:

```
struct Item {
  a: u32,
  b: Option<&'static str>,
  c: String,
}

let item = Item { a: 42, b: None, c: String::new("foobar") };
```

we would write

```
let item = ItemBuilder::new()
    .a(42)
    .c("foobar")
    .build();
```

We might not actually know the internal representation of `Item`, we just know the API of `ItemBuilder` and understand that we will get an `Item` after we call `build`. It is possible to provide some niceties like being able to pass a `&str` to the method for `c` instead of having to create the `String` explicitly, and using a default `None` value for `b` if it is not otherwise specified.

You might want to keep the internal representation of your struct hidden and therefore not expose the fields for direct construction, but you also want to make constructing an object easy. We have used builders in other chapters, for example when building a network request with the `reqwest` library, we use a `RequestBuilder`.

Some advanced uses of the builder pattern in Rust are to implement a form of what is sometimes called the typestate pattern. Specifically, as it is possible to return different types from each method, we can ensure that certain constraints are satisfied at compile-time. An example is best to illustrate this, although we will keep things a bit contrived to focus on the design pattern.

Suppose we want to build a network request:

```
struct Request {
  url: String,
  body: String,
  token: Option<String>,
}
```

We require the URL to be set before the body but the token can be set at any time if it is set at all. We can only build a request if we have both a URL and a body. We start with a `RequestBuilder`:

```
struct RequestBuilder {
  token: Option<String>,
}
```

The entry point for our API is `RequestBuilder::new()`, then we only provide the methods which are valid to call. In particular, there is no `body` method because the body must be set after the URL:

```
impl RequestBuilder {
  pub fn new() -> Self {
    RequestBuilder { token: None }
  }

  pub fn token(mut self, token: String) -> Self {
    self.token = Some(token);
    self
  }

  pub fn url(self, url: String) -> RequestWithUrlBuilder {
    RequestWithUrlBuilder {
      url,
      token: self.token,
    }
  }
}
```

We change the return type after the URL method because this moves us into a different conceptual state. This state is one where we know we have a URL:

```
struct RequestWithUrlBuilder {
  url: String,
  token: Option<String>,
}
```

Again we implement only the methods which are valid to be called in this state:

```
impl RequestWithUrlBuilder {
  pub fn token(mut self, token: String) -> Self {
    self.token = Some(token);
    self
  }

  pub fn body(self, body: String) -> FullRequestBuilder {
    FullRequestBuilder {
      url: self.url,
      body,
      token: self.token,
    }
  }
}
```

We move states to one with a known `url` and `body` by changing types again:

```
struct FullRequestBuilder {
  url: String,
  body: String,
  token: Option<String>,
}
```

Finally, we can expose a `build` method which will get us the final request type we are trying to produce:

```
impl FullRequestBuilder {
  pub fn token(mut self, token: String) -> Self {
    self.token = Some(token);
    self
  }

  pub fn build(self) -> Request {
    Request {
      url: self.url,
```

```
      body: self.body,
      token: self.token,
    }
  }
}
```

Given this API, we can build a request:

```
let req = RequestBuilder::new()
    .url("www.example.com")
    .body("foobar")
    .build();
```

and we can build one with a token:

```
let req = RequestBuilder::new()
    .url("www.example.com")
    .token("abc123")
    .body("foobar")
    .build();
```

Importantly, the following are compile-time errors:

```
RequestBuilder::new()
    .body("foo")
    .url("www.example.com")
    .build();

RequestBuilder::new()
    .url("www.example.com")
    .build();
```

This pattern is pretty simple in this context, but it is merely a specific example of a much broader technique of embedding logic into the type system.

However, this type of complex builder is not needed that often. Frequently, you just have many structs with many default fields and just need builders for each. In this case it is annoying to have to write basically the same code for each struct you want a builder for. It would be nice if you could just "derive" all of the builder code from the definition of your struct.

Initial setup

We are writing a procedural macro and as of today this requires its own standalone crate. This crate has a specific type which the compiler treats specially to get all this to work. Therefore, let's get started by having cargo create a new crate for us:

```
cargo new builder
```

We will add some dependencies to our manifest and also specify under the `lib` key that we are a `proc-macro`:

Cargo.toml

```
[package]
name = "builder"
version = "0.1.0"
authors = ["Your Name <your.name@example.com>"]
edition = "2018"

[lib]
proc-macro = true

[dependencies]
proc-macro2 = "1.0"
quote = "1.0"
syn = { version = "1.0", features = ["full"]}
```

The one `proc-macro = true` line is what tells the compiler to inject a specific dependency into your crate and also that your procedural macros that are exported can actually be used by other crates.

The compiler automatically exposes a library called `proc-macro` to your code as if it was a crate that you depend on. This is a bit quirky and has some negative consequences. Primarily the downside is that code that uses `proc-macro` can only execute in procedural macro contexts. This means that you cannot use tools for understanding Rust syntax in normal code unless it does not depend on `proc-macro`. Furthermore, you cannot unit test the `proc-macro` code.

A separate crate `proc-macro2` was created to fix these problems. The foundational crates for parsing Rust syntax, `syn`, and generating Rust syntax, `quote`, are written against the `proc-macro2` crate. Our basic plan is to write a macro with these tools, and then write some tiny amount of code to shuttle back and forth between the builtin `proc-macro` crate and `proc-macro2`.

Building the scaffold

Looking at the source of many popular procedural macro crates you might be surprised to discover they are often implemented as a single `lib.rs` file. This is not necessary, nor universal, but it often is the case because procedural macros tend to have very specific and limited scope.

You might want to ask if your macro really needs to be doing everything you have it doing if you find yourself breaking things up too much. On the other hand, if you are building something like serde[53] then you can benefit quite a bit from modularity.

Our macro will not have much configuration or other extra bits so a single `lib.rs` file works quite well. Let's get started with the imports that will make our lives easier:

[53] https://serde.rs/

src/lib.rs

```
extern crate proc_macro;
use proc_macro::TokenStream;
use quote::quote;
use std::fmt;
use syn::parenthesized;

use syn::parse::Result as SynResult;
```

The first line might be a surprise as the 2018 edition did away with the need for `extern crate` declarations. However, as of this writing because `proc_macro` is a builtin crate, you must still use the `extern crate` syntax. This is because the compiler looks in your manifest to resolve crate references but you can't put `proc_macro` as a normal dependency in the manifest because it isn't normal. Now you don't have to use `extern crate` with `std` but `proc_macro` did not get that special treatment. There is an unused `meta` crate that is automatically included that might replace `proc_macro` in the future, but not today.

Let's write the public interface to our library:

src/lib.rs

```
#[proc_macro_derive(Builder, attributes(builder))]
pub fn builder_derive(input: TokenStream) -> TokenStream {
    let ast = syn::parse(input).expect("Could not parse type to derive \
Builder for");

    impl_builder_macro(ast)
}
```

This is the entirety of what is exported by our library. Before turning to the overall structure of this function and what the attribute means, let's look at the implementation. We call `syn::parse` to turn the input into a data structure representing the AST (Abstract Syntax Tree) of the item we are defined on. We then pass that to our real implementation, `impl_builder_macro` which we will get to later. The type signature

of `impl_builder_macro` along with type inference is what tells `syn::parse` what type to turn our input into.

Here and throughout this chapter, the docs for the syn crate[54] will be an invaluable resource. The it can be a bit daunting to try to figure out what types are possible as parts of the syntax as you tear it down and build it back up. The docs are very thorough and can help give you a grasp on the complexity of Rust syntax.

This is the highest level of protection against syntax errors as we use `expect` to panic if we can't parse the input. It is common to panic or use the `compile_error` macro in procedural macros. This is because the code is executing at compile time so causing a panic is sometimes the only thing you can do to stop the compilation process when you cannot proceed. This is not runtime code so some of the normal rules do not apply.

As a further note along this line, we are operating very early in the compilation process. This is before type checking and before a lot of other compiler passes. We only have very basic information at the syntax level, we have no semantic information. For example, this means there is no way to say do something special if a type implements a trait. You are only give the literal tokens that make up the syntax of the source code. You can do a lot with this, but remember that this is nowhere near as much information as the full compilation process works with.

Let's break down the function a little bit more by looking at the `proc_macro_derive` attribute. This attribute takes the name of the derive as the first argument which is `Builder` in our case. This makes it so we can write:

```
#[derive(Builder)]
struct Request {
  ...
}
```

The second argument to `proc_macro_derive` is optional and defines helper attributes. The syntax here is to write `attributes(...)` with a comma separated list of attributes that we want to define inside the parentheses. We are defining a single attribute, `builder`, so that we can write:

[54] https://docs.rs/syn/1.0/syn/index.html

```
#[derive(Builder)]
struct Request {
  #[builder]
  pub x: i32,
  ...
}
```

We are going to use a slightly more complicated form of that attribute when we get around to defining how we will handle it. But to cover the general case here, attributes can take arguments if you want them to or not. So you might also see:

```
#[derive(Builder)]
struct Request {
  #[builder(foo, bar=32)]
  pub x: i32,
  ...
}
```

It is up to the implementation of the macro to decide what the attribute means and whether arguments mean anything. We will get the raw syntax and have to decide what to do with it. It might not be immediately obvious but even though we are declaring the name of an attribute here, all macros get access to all attributes. Therefore, other macros will see our attributes, they typically just choose to ignore them. You have to declare them like this to inform the compiler that such an attribute is not a syntax error.

The function `builder_derive` has the signature `(TokenStream) -> TokenStream` which is the form that every custom derive must implement. The input is the item that the derive is defined on and the return value is appended to the module or block where that item is defined. In other words,

```
#[derive(Builder)]
struct Request {
  #[builder(foo, bar=32)]
  pub x: i32,
  ...
}
```

effectively becomes

```
struct Request {
  pub x: i32,
  ...
}

// output from builder_derive(...) as syntax
```

Custom attribute macros have a signature of `(TokenStream, TokenStream) -> TokenStream` where the first argument is the arguments to the attribute itself and the second attribute is the item the attribute is on. The return value replaces the item with an arbitrary number of items. For example,

```
#[get("/")]
fn foobar() {
}
```

means that there is a procedural macro that defines a function `get` which will receive the token stream representing `("/")` as its first argument, and the token stream representing `fn foobar() {}` as its second argument. The token stream output from that function will replace all of that code.

Function-like procedural macros have the same signature as a derive macro, `(TokenStream) -> TokenStream`, where the input is the entirety of the macro invocation, but instead of getting appended to the module the token stream that is returned replaces the input at the same location in the source.

Besides using different attributes to mark the exported function that is about all the extra information you need to know to implement these other types of procedural macros after we work our way through a custom derive.

Let's get back to our implementation by writing the `impl_builder_macro` function we referenced earlier:

src/lib.rs

```
16 fn impl_builder_macro(ty: syn::DeriveInput) -> TokenStream {
17     match parse_builder_information(ty) {
18         Ok(info) => info.into(),
19         Err(e) => to_compile_errors(e).into(),
20     }
21 }
```

The point of this function is to move into the world of `proc_macro2` by passing the `syn` input into a function which only operates in this other world. We then use an `into` implementation of a type we will write soon to convert back into the `proc_macro` world which we then use to return the expected `TokenStream` back up. Note that the `proc_macro2::TokenStream` type implements the `Into` trait to get a `proc_macro::TokenStream` so we expect to get `proc_macro2::TokenStream` values that we just need to call `into` on.

Friendly error handling

We are also converting to the more standard style of using result style error propagation by handling the transformation of an error into something that the procedural macro system can work with. We do that by writing the `to_compile_errors` function that we will see next:

src/lib.rs

```
23 fn to_compile_errors(errors: Vec<syn::Error>) -> proc_macro2::TokenStre\
24 am {
25     let compile_errors = errors.iter().map(syn::Error::to_compile_error\
26 );
27     quote! { #(#compile_errors)* }
28 }
```

We assume that our errors come as a vector of `syn::Errors` which are the expected type of errors we will encounter. That is, we are going to mostly be running into syntax errors. One nice feature of `syn` is the associated function `syn::Error::to_compile_error` which converts the error type into a nice diagnostic error which the compiler will understand when returned as a token stream.

This is our first place where we are actually generating code. The `quote!` macro uses a syntax similar to the `macro_rules` macro for generating code, except it interpolates variables using the syntax `#variable`. This interpolation requires the variable to implement the `ToTokens` trait.

In our case we are interpolating the `compile_errors` variable. However this variable is an iterator. Therefore, like in declarative macros, we use the `#(...)*` syntax to generate code for each element in the `compile_errors` iterator.

The output of the `quote!` macro is the interpolated syntax as a `proc_macro2::TokenStream`. The inside of the macro in this case is just an interpolation, but it can be arbitrary Rust syntax. We will see this more later, but one nicety of procedural macros is that you can just write normal Rust with some interpolations rather than the more complicated syntax construction of a declarative macro.

As our error function expects a vector of errors we declare a type alias to make the corresponding Result type easier to write:

src/lib.rs

```
28 type MultiResult<T> = std::result::Result<T, Vec<syn::Error>>;
```

Furthermore, we define a struct to make working with a vector of errors a little easier:

src/lib.rs

```
52 #[derive(Debug, Default)]
53 struct SyntaxErrors {
54     inner: Vec<syn::Error>,
55 }
```

The advantage of doing this is to implement the following helper functions:

src/lib.rs

```
57 impl SyntaxErrors {
58     fn add<D, T>(&mut self, tts: T, description: D)
59     where
60         D: fmt::Display,
61         T: quote::ToTokens,
62     {
63         self.inner.push(syn::Error::new_spanned(tts, description));
64     }
65
66     fn extend(&mut self, errors: Vec<syn::Error>) {
67         self.inner.extend(errors);
68     }
69
70     fn finish(self) -> MultiResult<()> {
71         if self.inner.is_empty() {
72             Ok(())
73         } else {
74             Err(self.inner)
75         }
76     }
77 }
```

We create an `add` method which appends a single error to our vector. This uses generic types to accept anything that can be turned into tokens along with anything that can be nicely printed as the description. The `new_spanned` function uses the token trees, `tts`, input to capture source information to information the compiler where to draw errors when printing the error out.

A span in the Rust compiler is effectively a region of source code. You can think of it as a start and end position used to draw the arrows and lines in the compiler error messages. It can get more complicated than that as you might know if you have ever seen the nice diagrams that come out with borrow checker errors. But for our purposes you can think of it has a region of source code. Each piece of syntax defines a span so usually you can bootstrap a span if you have some input tokens.

Getting this right is sometimes a bit of an art and can really make your macro maddeningly hard to use or delightful. The goal is to inform the compiler as much as possible as to what syntax is causing the problem and to describe as best as you can how to fix it. The worst scenario is when the error just points at `#[derive(Builder)]` and has some opaque message.

The `extend` method is self-explanatory. The `finish` method consumes this wrapper struct to return a value of our `MultiResult`. The consequence of this is that you can use the `?` operator after calling finish to report as many errors as you can diagnose at once. This is another nice feature for your users. It is great if you can discover multiple errors at once so they can fix them all rather than having to fix one only to find out there are more lurking under the covers. Sometimes it is not possible or desirable to find all errors up front, but when you can you should. This vector of errors makes it possible.

Getting into the parser

Okay, we have our error infrastructure in place. Now we can write our next helper function which takes the `syn` input and returns a result with our builder code or a vector of errors:

src/lib.rs

```
30 fn parse_builder_information(ty: syn::DeriveInput) -> MultiResult<Build\
31 erInfo> {
32     use syn::spanned::Spanned;
33     use syn::Data;
34
35     let span = ty.span();
36     let syn::DeriveInput {
37         ident,
38         generics,
39         data,
40         attrs,
41         ..
42     } = ty;
43
```

```
    match data {
        Data::Struct(struct_) => parse_builder_struct(struct_, ident, g\
enerics, attrs, span),
        _ => Err(vec![syn::Error::new(
            span,
            "Can only derive `Builder` for a struct",
        )]),
    }
}
```

The first line brings the `Spanned` trait into scope so that we can call the `span` method on our input. We destructure the `syn::DeriveInput` type into the specific constituent parts that we care about. Specifically, we match the data field against the `syn::Data` enum to see if the item we are defined on is a struct.

The entire purpose of this function is to ensure that we are being derived on a struct and not on an enum or any other possible item. If we are not being derived on a struct then we create an error and stop here. Otherwise, we pass the pieces of data that we have gathered on to yet another helper function.

Let's turn now to the `BuilderInfo` type we see in our type signature:

src/lib.rs

```
struct BuilderInfo {
    name: syn::Ident,
    generics: syn::Generics,
    fields: Vec<(Option<syn::Ident>, syn::Type, Vec<BuilderAttribute>)>,
}
```

This is all of the information we need from the parsed AST to be able to generate the code we want. We have the name of the type as an `syn::Ident`, we have the declaration of any generics from the type, and lastly we have a vector of fields. We will get to `BuilderAttribute` next, but as an example consider:

```
#[derive(Builder)]
struct Item<T, U>
where
  T: Default
{
  a: u32,
  b: T,
  #[builder(required)]
  c: U
}
```

Here name is `Item`, the generics field captures `<T, U> where T: Default`, and fields contains `a: u32`, `b: T`, and `#[builder(required)] c: U`. Each of those are wrapped in a suitable data structure that both captures this syntax as well as information about where it lives in the source. The fields vector contains a tuple of identifier, type, and attributes so for example `a: u32` would be something like `(Some(a), u32, vec![])`. We will see how this exactly plays out later, but this should give you an idea of what this structure holds.

Handling attributes

We turn now to `BuilderAttribute` which is an enum defining all of the attributes we support. We are only going to support one variant but this should elucidate how you can manage multiple attributes.

src/lib.rs

```
247 enum BuilderAttribute {
248     Required(proc_macro2::TokenStream),
249 }
```

This is going to capture the attribute:

```
#[builder(required)]
```

The meaning of this attribute is to specify that a field must be set as part of the build process and therefore a default value should not be used. For simplicity, we are going to enforce this by causing a panic if the field is not set. For fields not marked required, we will assume the type of the field implements the `Default` trait. For a different use case you might want to flip this assumption and make everything required and only make things default if they are explicitly marked. Or you could do something more complicated that combines these two extremes.

As we said we only have one attribute that we care about, but we are going to go through the trouble of handling a list of attributes for explanatory purposes. Thus, we define `BuilderAttributeBody` to be a collection of `BuilderAttributes`. We then implement the `Parse` trait from `syn` for this type. Implementing this trait is how to work custom logic into the work that `syn` does.

src/lib.rs

```
268 struct BuilderAttributeBody(Vec<BuilderAttribute>);
269
270 impl syn::parse::Parse for BuilderAttributeBody {
271     fn parse(input: syn::parse::ParseStream) -> SynResult<Self> {
272         use syn::punctuated::Punctuated;
273         use syn::token::Comma;
274
275         let inside;
276         parenthesized!(inside in input);
277
278         let parse_comma_list = Punctuated::<BuilderAttribute, Comma>::p\
279 arse_terminated;
280         let list = parse_comma_list(&inside)?;
281
282         Ok(BuilderAttributeBody(
283             list.into_pairs().map(|p| p.into_value()).collect(),
284         ))
285     }
286 }
```

The purpose of our implementation of `parse` is to remove the parentheses from

`#[builder(...)]` so that `BuilderAttribute` only has to deal with the tokens inside. We also deal with the logic of a comma separated list here.

The `parenthesized!(inside in input)` means to take the `ParseStream` in `input`, remove the parentheses from the outside and store the inner tokens in `inside`. This is a macro defined in `syn` for this very common scenario. There are similar parser macros for removing curly braces (`braced!`) and square brackets (`bracketed!`).

The next step is to parse a sequence of `BuilderAttribute` types separated by commas allowing an optional trailing comma. The `Punctuated<T,P>` type is used to handle this very common case of a list of `T` separated by `P`. We use `parse_terminated` which allows trailing punctuation. If you do not want to accept trailing punctuation, then you can use `parse_separated_nonempty`. Comparing this to the declarative macro method of handling this is a good example of how different procedural and declarative macros are. We are writing real Rust code here and using types and traits to drive our implementation. This is in contrast to the Scheme style pattern matching of declarative macros.

The final work that needs to be done is to extract the `BuilderAttribute` types from this punctuated list. We do that by using the `into_pairs` method on `Punctuated` which returns an iterator of `Pair<BuilderAttribute, Comma>` and then for each of these we just call `into_value` to get the `BuilderAttribute` out. Finally, we call `collect` on the iterator to turn it into the vector that our return type expects.

This might seem a bit confusing, it again helps to read the docs for the `syn` crate for the `Punctuated` type. It is the only way that any of this can make sense. The common parsing tasks have specialized functions available so it is best to start there to hopefully find what you are looking for.

Okay, now we can turn to implementing the `Parse` trait for `BuilderAttribute`:

src/lib.rs

```
impl syn::parse::Parse for BuilderAttribute {
    fn parse(input: syn::parse::ParseStream) -> SynResult<Self> {
        use syn::Ident;

        let input_tts = input.cursor().token_stream();
        let name: Ident = input.parse()?;
        if name == "required" {
            Ok(BuilderAttribute::Required(input_tts))
        } else {
            Err(syn::Error::new(
                name.span(),
                "expected `required`",
            ))
        }
    }
}
```

The spirit here is quite simple, but the mechanics are little cumbersome. We want to check if the attribute is literally `required`, if so then we return a success, otherwise we declare a failure. We call methods from `syn` to turn the `input` into an `Ident` which is the only thing we expect to find. If this step fails then we will return an error because of the `?` operator. We then compare this `Ident` to `"required"`. If we get a match, then we wrap the input token stream inside our enum variant. Otherwise we use the location of the `Ident` that we did parse to generate an error saying that we got something unexpected.

With all of these parse functions in place, we can write a helper to go from a vector of `syn::Attributes` to our desired type:

src/lib.rs

```
fn attributes_from_syn(attrs: Vec<syn::Attribute>) -> MultiResult<Vec<B\
uilderAttribute>> {
    use syn::parse2;

    let mut ours = Vec::new();
    let mut errs = Vec::new();

    let parsed_attrs = attrs.into_iter().filter_map(|attr| {
        if attr.path.is_ident("builder") {
            Some(parse2::<BuilderAttributeBody>(attr.tokens).map(|body|\
 body.0))
        } else {
            None
        }
    });

    for attr in parsed_attrs {
        match attr {
            Ok(v) => ours.extend(v),
            Err(e) => errs.push(e),
        }
    }

    if errs.is_empty() {
        Ok(ours)
    } else {
        Err(errs)
    }
}
```

This is really just some boilerplate around the parsing functions we just wrote. We define a vector of attributes to return and a vector of errors. We will only return one of these in the end, but it depends on what we see along the way.

The `Iterator` trait has many useful methods, `filter_map` which we use here is a way to both map over an iterator and remove some unwanted items at the same time. The closure passed to `filter_map` returns an `Option<T>`. The resulting iterator will "unwrap" the values that are `Some(T)` to just be a `T` and will not include the `None` values. Due to how `Option` implements `IntoIterator`, you can also use `flat_map` here for the same effect, but `filter_map` is more descriptive as to what we are trying to accomplish.

We take the attributes we are passed as input, we ignore any that are not the `builder` attribute, and ones that do match we parse into our specialized types. The `parse2` function is for parsing `proc_macro2` types but is otherwise the same as `parse` for `proc_macro` types. Suppose we have:

```
#[something(else), builder(required)]
```

We would then be iterating over `something(else)` and `builder(required)`. The first thing we would see is an `attr.path` of `something` which is not `builder` so we return `None` for that attribute which effectively excludes it from the `parsed_attrs` result. The next thing we see has a path that matches `builder` so we take then tokens of the attribute which is `(required)` and we parse that into a `BuilderAttributeBody` which relies on our `Parse` trait implementation from earlier. Once we have that we call `map(|body| body.0)` which is because the `Parse` trait returns a `Result` so we have to deal with getting inside the `Ok` variant to pull the `Vec<BuilderAttribute>` out of the tuple struct wrapper we put around it.

We had to do it this way because we could not implement `Parse` on `Vec<BuilderAttribute>` directly as we do not own `Vec` nor `Parse`. Rust trait implementation rules require that you either own the trait or own the type (where own means is defined within the crate with the trait implementation). We do own our tuple struct so we can implement `Parse`. But then to get what we want out we have to do this little map trick to pull the vector out.

The rest of the function is straight forward. We iterate over the `parsed_attrs` and accumulate the good and bad parts. Then if we encountered no errors we can return our good result, otherwise we return the errors.

Finishing up the parsing

Now we can turn to the real meat of our parsing. Let's start with the signature:

src/lib.rs

```
fn parse_builder_struct(
    struct_: syn::DataStruct,
    name: syn::Ident,
    generics: syn::Generics,
    attrs: Vec<syn::Attribute>,
    span: proc_macro2::Span,
) -> MultiResult<BuilderInfo> {
```

We already dealt with ensuring that we are getting derived on a struct so we know that we have a `syn::DataStruct` to work with. The rest of the input was pulled out of the parsed input because it is the only things we need to eventually define a `BuilderInfo` struct. The point of this function is to deal with all of the various error cases that might occur so that we know if we end up with a `BuilderInfo` struct that everything is legit for doing code generation.

The first step is to check the attributes defined on the struct itself to see if anyone tried to use a `builder` attribute there:

src/lib.rs

```
    use syn::Fields;

    let mut errors = SyntaxErrors::default();

    for attr in attributes_from_syn(attrs)? {
        match attr {
            BuilderAttribute::Required(tts) => {
                errors.add(tts, "required is only valid on a field");
            }
        }
    }
```

We do not support `#[builder(required)]` on the entire struct so we add an error to our collection of errors if we see one. We do not return early here as we want to keep parsing and collecting errors if there are more that we can find.

src/lib.rs

```
186     let fields = match struct_.fields {
187         Fields::Named(fields) => fields,
188         _ => {
189             errors.extend(vec![syn::Error::new(
190                 span,
191                 "only named fields are supported",
192             )]);
193             return Err(errors
194                 .finish()
195                 .expect_err("just added an error so there should be one\
196 "));
197         }
198     };
```

Our next step is to get a handle on the fields defined on the struct. You can define unnamed struct fields if you are defining a tuple struct, for instance

```
struct Foo(String)
```

has one unnamed field. We do not support that type of struct. We therefore look for named fields and pull out the inner data, otherwise we add an error and then return all the errors we have gathered so far. We do not go any further looking for errors because without named fields there is not much we can do.

src/lib.rs

```
198     let fields = fields
199         .named
200         .into_iter()
201         .map(|f| match attributes_from_syn(f.attrs) {
202             Ok(attrs) => (f.ident, f.ty, attrs),
203             Err(e) => {
204                 errors.extend(e);
205                 (f.ident, f.ty, vec![])
206             }
207         })
208         .collect();
```

For each of our named fields, we need to extract the identifier, type, and attributes. We do this by iterating over the fields and then using methods on the field type to get the information we want. We use our previously defined `attributes_from_syn` to extract attribute information. Note that we will look at the attributes for every field so we have the potential to accumulate multiple errors depending on the input.

Finally, we return our errors if we encountered any, or we return a successful result containing our `BuilderInfo` struct:

src/lib.rs

```
210     errors.finish()?;
211
212     Ok(BuilderInfo {
213         name,
214         generics,
215         fields,
216     })
217 }
```

Generating code

All of the work so far has been on the parsing side of the macro. We rely very heavily on `syn` to do the hard work of parsing Rust syntax but it still takes quite a bit of code

to extract the necessary information out of the extensive information that is available to us. But now we are done parsing in the sense that we can move forward under the assumption of a well defined `BuilderInfo` struct that we just need to generate code for.

Our first step is to get an implementation of `Into<TokenStream>` for `BuilderInfo`:

src/lib.rs

```
85 impl From<BuilderInfo> for TokenStream {
86     fn from(other: BuilderInfo) -> TokenStream {
87         other.generate_builder().into()
88     }
89 }
```

As we have seen before we choose to implement `From<BuilderInfo>` for `TokenStream` rather than implementing `Into` directly and then rely on the automatic reflexive implementation. This is just yet another indirection to a helper method `generate_builder` defined on `BuilderInfo`. This is a stylistic choice, but it can be useful to keep your trait implementations slim and keep the logic that really works with the internals of your struct inside an `impl` block for the struct itself.

The other advantage here is that, as we will see shortly, `generate_builder` can operate only in terms of `proc_macro2` types and this trait encapsulates the `proc_macro2` to `proc_macro` dance using another call to `into`.

Let's get started with our code generation by getting the preamble out of the way:

src/lib.rs

```
91 impl BuilderInfo {
92     fn generate_builder(self) -> proc_macro2::TokenStream {
```

We are inside the `impl` block for our type and we are writing a function which consumes the struct to return a `proc_macro2::TokenStream`. There is nothing else happening after this function so there is no reason not to consume self here.

The structure of the implementation of this function is a little bit inside-out so it might help to go through quickly to the end and then come back to clarify. This is

because we create variables that hold bits of code which then get interpolated into the final code.

First, we build the code for the setters:

src/lib.rs

```
93          let gen_typ = syn::Ident::new("__Builder_T", proc_macro2::Span:\
94  :call_site());
95
96          let setters = self.fields.iter().map(|(n, t, _)| {
97              quote! {
98                  fn #n<#gen_typ: Into<#t>>(mut self, val: #gen_typ) -> S\
99  elf {
100                     self.#n = Some(val.into());
101                     self
102                 }
103             }
104         });
```

For each field we create a function that can be used to set the value for that field. The `quote!` macro allows us to write normal Rust code and interpolate variables in scope with `#variable`. So it is easiest to understand the generated code by looking at an example. If we have a field like `field_name: U`, then this code creates the following code:

```
fn field_name<__Builder_T: Into<U>>(mut self, val: __Builder_T) -> Self\
 {
  self.field_name = Some(val.into());
  self
}
```

The one weird thing here is that we create an identifier `__Builder_T` which we use as our generic type variable. This gets into the concept of macro hygiene which can be a complicated topic. The basic problem we are trying to work around is suppose we used `T` as the type variable but `T` was already defined to be something in the surrounding code.

One type of hygiene would allow that and would treat those two `T` types has different because they were created in different contexts. However that does not allow you create identifiers that you want to be able to be referred to outside of your macro.

A different type of hygiene makes identifiers that you create be the same as if they were part of the source where the macro is called. This is sometimes called unhygienic. This means your created identifiers can be used by regular code. However this also means they can accidentally conflict. You must specify the type of hygiene you are opting into when creating an identifier by passing a `syn::Span` to the constructor of an identifier. We use `call_site` hygiene here to show what it is. This is the type that would otherwise conflict with an existing name, but it is also the type we have to use later. We therefore use `__Builder_T` as it is highly unlikely this will conflict with an existing type. You could alternatively use `def_site` hygiene here to avoid this.

The next thing we create are the fields on the builder struct itself:

src/lib.rs

```
104         let builder_fields = self.fields.iter().map(|(n, t, _)| {
105             quote! {
106                 #n: Option<#t>,
107             }
108         });
```

The builder struct holds the state as we are in the process of building up the data. We choose to implement this by having an optional field for each field in our target struct. That is, if we have the struct:

```
#[derive(Builder)]
struct Item {
  a: u32,
  b: String,
}
```

then we will have a builder struct like:

```
struct ItemBuilder {
  a: Option<u32>,
  b: Option<String>,
}
```

This code takes the field `a: u32` and generates the field:

```
a: Option<u32>,
```

It is just that snippet of that we are generating. Obviously this makes no sense without being inside a larger struct definition, but we need this individual pieces to be able to build up that larger structure.

The next thing we build is a default field for each field:

src/lib.rs

```
110         let builder_defaults = self.fields.iter().map(|(n, _, _)| {
111             quote! {
112                 #n: None,
113             }
114         });
```

This will be used to implement `Default` for our builder struct which we will do by setting every field to `None`.

The next piece is the code for each field in our build function:

src/lib.rs

```
116         let builder_build = self.fields.iter().map(|(n, _t, a)| {
117             if a.is_empty() {
118                 quote! {
119                     #n: self.#n.unwrap_or_else(Default::default),
120                 }
121             } else {
122                 quote! {
123                     #n: self.#n.unwrap(),
124                 }
125             }
126         });
```

We know that our attribute vector will either be empty or have one element. If it has one element then the field is required, otherwise it might never be set. In the case that it is not required we use `Default::default` to fill in the value. Therefore, if a field does not implement `Default` it must be marked required. We use `unwrap` on the option for required field which will cause a runtime panic if build is called when a required field is not set. We leave it up to you to extend this to more complex error handling scenarios. Again we are just defining a snippet of the whole which makes no sense without being inside something larger. That is, this code takes a field such as `a: u32` and generates:

```
a: self.a.unwrap(),
```

The last pieces we need before putting everything together are the name of the builder and the generics information:

src/lib.rs

```
128         let name = self.name;
129         let (impl_generics, ty_generics, maybe_where) = self.generics.s\
130 plit_for_impl();
131         let builder_name = syn::Ident::new(&format!("{}Builder", name),\
132  name.span());
```

We take the `name` which is the identifier of the struct, that is for

```
1  #[derive(Builder)]
2  struct Item {
3    ...
4  }
```

the name is `Item` and we construct an identifier `ItemBuilder` which has hygiene as if it was in the same context as where `Item` is defined. This is so that code that uses `Item` can use `ItemBuilder` as if it was hand-written in the same place.

The function `split_for_impl` on the generics type is defined exactly to give you the pieces of the generic information you need so that they can be interpolated as you generate code. The generics defined on a struct need to be put in different places when you define a trait for that struct.

We can finally output the code which defines our builder struct:

src/lib.rs

```
131         quote! {
132             impl #impl_generics #name #ty_generics #maybe_where {
133                 fn builder() -> #builder_name #ty_generics {
134                     #builder_name::new()
135                 }
136             }
137 
138             impl #impl_generics Default for #builder_name #ty_generics \
139 #maybe_where {
140                 fn default() -> Self {
```

```
                    #builder_name {
                        #(#builder_defaults)*
                    }
                }
            }

            struct #builder_name #ty_generics #maybe_where {
                #(#builder_fields)*
            }

            impl #impl_generics #builder_name #ty_generics #maybe_where\
 {
                fn new() -> Self {
                    Default::default()
                }

                #(#setters)*

                fn build(self) -> #name #ty_generics {
                    #name {
                        #(#builder_build)*
                    }
                }
            }
        }
    }
}
```

The first part of this code adds an `impl` block for our struct which defines a function `builder` which returns an instance of our builder. If your struct already has a `builder` method then this will be a compiler error. This is one of the nice things in Rust about being able to define multiple `impl` blocks for the same item. Even if you have an `impl` for your struct, we can create another one and it just adds our functions.

The next part of this code implements `Default` for our builder. It does this by creating a struct literal by interpolating the iterator of fields defined by `builder_defaults`

using the `#(...)*` syntax we saw earlier.

Then we create the struct itself again using the repetition syntax `#(builder_fields)*` to interpolate the fields we defined earlier into the struct definition.

Finally, we create an `impl` block for the builder struct which defines `new` to return `Default::default`, interpolates all of the setter functions, and then defines a `build` function which consumes the builder and constructs an instance of the struct we are trying to build. This uses `builder_build` fields we created above.

If we squint and ignore all of the noise from the `#` characters and imagine the `#(...)*` as multiple values the structure of the code should hopefully look like what you expect. If not then it might help to come back here after we go through a full example of using this derive and the code it generates.

Using our custom derive

Let's take this bad boy for a spin. Create a new crate with whatever name you like:

```
cargo new builder-test
```

We edit our manifest to depend on our procedural macro crate:

Cargo.toml

```
[package]
name = "builder-test"
version = "0.1.0"
authors = ["Your Name <your.name@example.com>"]
edition = "2018"

[dependencies]
builder = { path = "../builder", version = "0.1.0" }
```

I created this test crate in a sibling directory to the macro crate so that we can use this simple `path` specifier to rely on the local crate that we just created. The path is relative to the `Cargo.toml` file which is why it requires going up one level.

Let's put some code in `main.rs` to get started:

src/main.rs

```
use builder::Builder;

#[derive(Debug)]
struct X {}

#[derive(Debug, Builder)]
struct Item<T, U>
where
    T: Default,
{
    a: u32,
    b: Option<&'static str>,
    c: String,
    #[builder(required)]
    d: X,
    e: T,
    #[builder(required)]
    f: U,
}
```

We import the `Builder` macro from our crate and then define a couple structs to work with. We are just trying to stretch some of the complexity we handled by introducing generics and the `X` and `U` types which specifically do not implement `Default`.

Then in our `main` function, we can build some `Item` structs using the generated code:

src/main.rs

```
fn main() {
    let item: Item<i32, &str> = Item::builder()
        .a(42u32)
        .b("hello")
        .c("boom".to_owned())
        .d(X {})
        .e(42i32)
        .f("hello")
        .build();

    println!("{:#?}", item);

    let item2 = Item::<u32, u64>::builder().b(None).d(X {}).f(99u64).bu\
ild();
    println!("{:#?}", item2);
}
```

If we `cargo run` everything should compile and you should see the two items printed out. That at least means the builder works as intended.

To understand the generated code we can use `cargo expand` again to see what is happening. This will produce quite a bit of output, but the relevant bits for us are:

```
impl<T, U> Item<T, U>
where
    T: Default,
{
    fn builder() -> ItemBuilder<T, U> {
        ItemBuilder::new()
    }
}
impl<T, U> Default for ItemBuilder<T, U>
where
    T: Default,
```

```
{
    fn default() -> Self {
        ItemBuilder {
            a: None,
            b: None,
            c: None,
            d: None,
            e: None,
            f: None,
        }
    }
}
struct ItemBuilder<T, U>
where
    T: Default,
{
    a: Option<u32>,
    b: Option<Option<&'static str>>,
    c: Option<String>,
    d: Option<X>,
    e: Option<T>,
    f: Option<U>,
}
impl<T, U> ItemBuilder<T, U>
where
    T: Default,
{
    fn new() -> Self {
        Default::default()
    }
    fn a<__Builder_T: Into<u32>>(mut self, val: __Builder_T) -> Self {
        self.a = Some(val.into());
        self
    }
    fn b<__Builder_T: Into<Option<&'static str>>>(mut self, val: __Buil\
der_T) -> Self {
```

```
        self.b = Some(val.into());
        self
    }
    fn c<__Builder_T: Into<String>>(mut self, val: __Builder_T) -> Self\
 {
        self.c = Some(val.into());
        self
    }
    fn d<__Builder_T: Into<X>>(mut self, val: __Builder_T) -> Self {
        self.d = Some(val.into());
        self
    }
    fn e<__Builder_T: Into<T>>(mut self, val: __Builder_T) -> Self {
        self.e = Some(val.into());
        self
    }
    fn f<__Builder_T: Into<U>>(mut self, val: __Builder_T) -> Self {
        self.f = Some(val.into());
        self
    }
    fn build(self) -> Item<T, U> {
        Item {
            a: self.a.unwrap_or_else(Default::default),
            b: self.b.unwrap_or_else(Default::default),
            c: self.c.unwrap_or_else(Default::default),
            d: self.d.unwrap(),
            e: self.e.unwrap_or_else(Default::default),
            f: self.f.unwrap(),
        }
    }
}
```

Comparing this to the code to generate the code should clarify what each of the parts is doing.

Wrapping up

Macros are a powerful and incredibly useful language feature. Quite a bit of the positive ergonomics of using Rust comes from the ability to automatically derive traits. Hopefully you have quite a bit more insight into how this is done and how you can accomplish this if needed.

Macros are more than just for deriving traits and they can be a nice way to simplify many repetitive programming tasks. However, they can also be confusing, brittle, and overly complex for many tasks. Therefore, try not to use them, but know they are in your back pocket when you really need to reach for them.

Changelog

Revision 5 (02-20-2020)

- Fixes two code imports in the WASM chapter

Revision 4 (02-19-2020)

- Added Chapter 7: Macros
- Updates to the intro
- Fixed typos
- Chapter 3: Fixed the `messages-actix` code
- Updated `blog-actix` version
- Added `builder` and `builder-test` code

Revision 3 (01-29-2020)

Pre-release revision 3

Revision 2 (11-25-2019)

Pre-release revision 2

Revision 1 (10-29-2019)

Initial pre-release version of the book