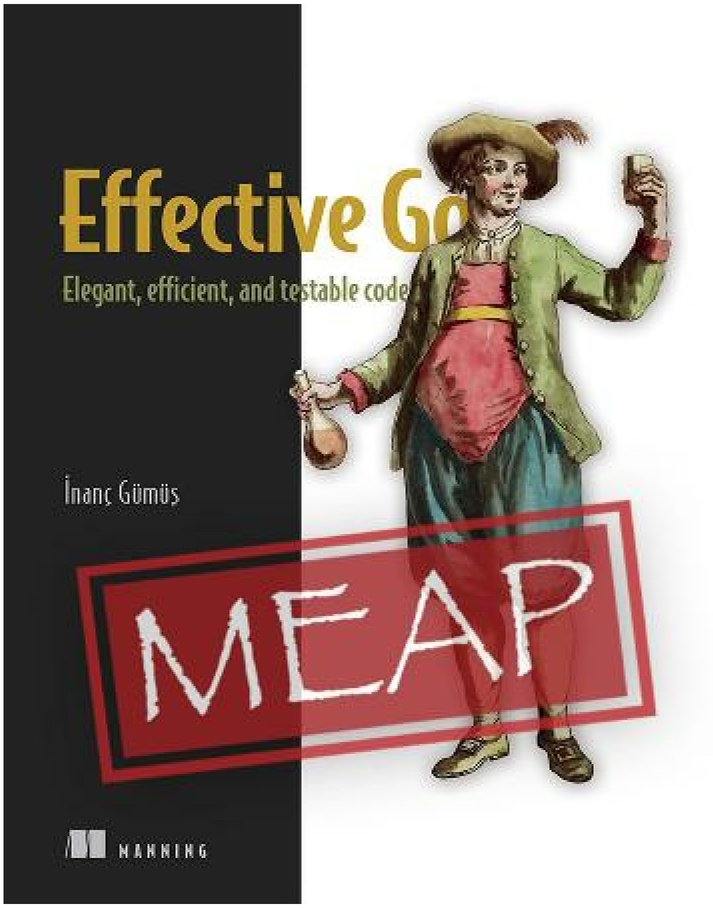
Effective Go

Elegant, efficient, and testable code

İnanç Gümüş

MEAP

MANNING


Effective Go
Elegant, efficient, and testable code
İnanç Gümüş
MEAP
MANNING

Effective Go MEAP V05

MEAP Edition Manning Early Access Program Effective Go Elegant, efficient, and testable code Version 5

For more information on this and other Manning titles go to

manning.com

welcome

Thanks for purchasing the MEAP for *Effective Go: Elegant, efficient, and testable code*. Well-designed code in Go demands a different mindset, and if you want to write Go code that's easy to test and maintain, you've come to the right place!

To get the most benefit from this book, you'll need to be a programmer, at least with an intermediate knowledge of any programming language. You may be new to Go, but knowing the basics will help along the way. If you don't know the basics of the Go programming language, you can learn it through my open-source repository that contains 1000+ explanations and exercises.

When I first started experimenting with Go five years ago, it was out of necessity. I needed a modern and simple programming language. Since that time, there have been numerous examples, some really great introductions to the language, and a few cookbooks to help newcomers learn the basics. As I write this, there's no book yet on writing and structuring practical and testable code in Go, but there's a growing need for developers. *Effective Go* is my attempt to write and maintain practical and testable code using Go.

Go is relatively a young language, and hundreds of thousands of developers from other programming languages come to Go every year. If you're one of them, you may be wondering how to transfer your existing knowledge of other languages, particularly when it comes to writing idiomatic and testable code in Go.

This book will teach you how to implement well-designed, testable, and practical Go projects from scratch. I'll be showing you different approaches—what to do, what not to do, and when and why you should use a particular method using practical examples you might encounter in the workplace.

I got the idea for this book based on questions I received from folks who've' taken my online course, my YouTube channel, or are following my blog.

Effective Go will answer these questions and more:

- How to write idiomatic and testable code in Go?
- How can I build an easy-to-maintain Go program from scratch?
- How can I test my program? What should I test—or not test? Am I testing correctly? What are the best practices? What are the anti-patterns?
- How can I organize my code?

The first part of the book is ready, and you'll learn the following:

- What makes Go and other programming languages different?
- Introduction to some of the primary features of Go: Type system, interfaces, embedding, inheritance vs. composition, concurrency vs. parallelism, goroutines, and channels.
- Writing an idiomatic and testable URL parser library package in Go. You'll learn about testing basics, writing maintainable tests, table-driven tests, subtests, always up-to-date executable documentation using example tests, internal tests, and more.

All said I hope you enjoy this book and that you find it helpful to your own programming practice. If you have any questions, comments, or suggestions, please share them in Manning's liveBook Discussion forum for my book. And, if you want to stay up to date on all the latest Go test best practices, be sure to give my Twitter a follow.

Thank you!

— Inanc Gumus

In this book

1 Warming up

This chapter covers

- Goals of the book
- Importance of code design and testability
- Bird-eye's view introduction to Go

This chapter will show you what Go offers from a bird's eye view and how it is different from other languages. This chapter will be an introduction and won't cover everything in detail, but it will give you an overview.

Throughout the book, I will help you design and test maintainable code in Go. Go is a tiny and straightforward language. But don't let that fool you; you are standing on the shoulders of giants before you. To achieve simplicity, a lot of thought went into its design mechanics. Behind the scenes, Go is a complex language with many moving parts, but it deliberately hides that complexity behind simple interfaces.

1.1 Goals of the book

I wrote this book for intermediate-level programmers who have recently learned how to program with Go. Go has distinct design idioms and mechanics and will bite you if you fight with the language and write code as you do in other programming languages. So you cannot write well-designed code without a proper understanding of how Go approaches program design.

Let's imagine you're a Java programmer. You heard that Go is fast, reliable, cross-platform, comes with great tooling, out-of-the-box libraries, and is easy to learn and use. So, you decided to port one of the command-line tools you had written from Java to Go. Just the correct type of person for this book!

After reading a couple of tutorials and some hard work, you managed to write the program in Go. You decided to add one more feature to your program. But it seems more tricky than you expected. You stumble upon a problem, and soon after another. You observe that the code is increasingly becoming an unmaintainable mess.

You already know how to design Java code. But, now, almost everything you know suddenly stopped helping you. You feel lost and frustrated. Your instincts tell you something is missing. You're starting to think that you might not have well-designed code that will endure the test of time. You realize you can't figure out how to design a maintainable program. You begin reading books and online articles. After some digging, you realize that Go might have a different perspective on program design. Sounds familiar? This book enters the scene right there and will teach you to write well-designed, testable, and maintainable code in Go.

1.1.1 Well-Designed code

Let's talk about what you are going to learn in the book. The primary goal of this book is to write well-designed and maintainable code in Go. *Well-designed code is simple, and simple is not easy*. Well-designed code is easy to reason about, easy to change, reliable, less buggy, and helps you avoid surprises.

Since the words *well* and *design* can mean different things depending on the context, there is no single truth but guidelines. Some people can look at the same code and think: "*Oh, this is terrible!*" while others may think: "*Woah, this is awesome!*". In the end, it's about creating code that can survive by quickly adapting to the changing needs.

Creating well-designed code in Go only gets easier after fully understanding the language mechanics. Without that knowledge, you will probably fight with the language and bring your previous design decisions from other programming languages. The good news is that there is usually one right way to do things in Go which we call *Idiomatic Go*. Here are some of the qualities that we expect to see from good code:

- `Simple`–Code is straightforward, easy to read, and understandable. There is no magic in Go code: You can understand the hardware cost of almost everything you do.

- `Adaptable`–Code is easily adaptable to changing requirements. Go follows the Unix philosophy and design with `composability` in mind instead of `inheriting` behavior from other types.
- `Testable`–Code is straightforward to test.

Of course, there will be plenty of others throughout the book. That's why I wrote this book! But I think these properties can give you a good mindset of what we are trying to achieve in Go.

1.1.2 Testable code

"Nothing endures but change."

— Heraclitus

The other thing that you will learn in the book is writing testable code. Fortunately, testing is a first-class citizen in Go and the Go Standard Library has good support for it.

One of the most critical strengths of the software development industry is that the software can change. Code should adapt to new requirements and hopefully pass the test of time. Yet, software that is stubborn to change doesn't have this advantage. Crafting such code may get easier to achieve with tests. Without tests, you can't sensibly be sure whether your code still works after you make a change.

Back in the 90s, I was manually testing my code. I was writing some code and then running it to see if it was doing what I expected. But trying to verify software manually in this way is error-prone and unscalable. Especially in a large codebase, it becomes impossible to test things manually. Fortunately, since the early 2000s, I test my code with both manual and automated testing. But even with automated tests, it is still impossible to develop an entirely error-free program. Who has an endless amount of time to create every test case out there?

Although tests can help you find bugs, it's only half of the story. Testing is also about creating testable code, and doing so is an art in itself. There isn't a single truth set in stone for every use case out there. Creating testable code may also improve your software design skills. When crafted correctly, testable code can help you design robust, adaptable, and healthy programs. When you write tests, you will be exercising your code from the eyes of your tests. This is important because it lets you discover first-hand how easy or hard it is to use your code.

Let's stop for a moment and take a look at some benefits of testing. When you craft good enough tests, there are many benefits. I will explain what makes a test good enough in detail later on. Here are some of them:

- *Confidence*—You will trust your code and improve it without fear.
- *Modular design*—Tests can help you craft a high-quality codebase with good design traits like decoupling and high cohesion.
- *Fewer bugs*—Research has proved that testing tremendously reduces and finds bugs early on.
- *Debugging*—Instead of manually finding bugs, tests can help you automatically find them whenever you change something.
- *Documentation*—Go has first-class support for documenting your code and even enforces it. Tests can become ever-updating documentation for your code. When I'm trying to understand a piece of software, I always read the tests first.

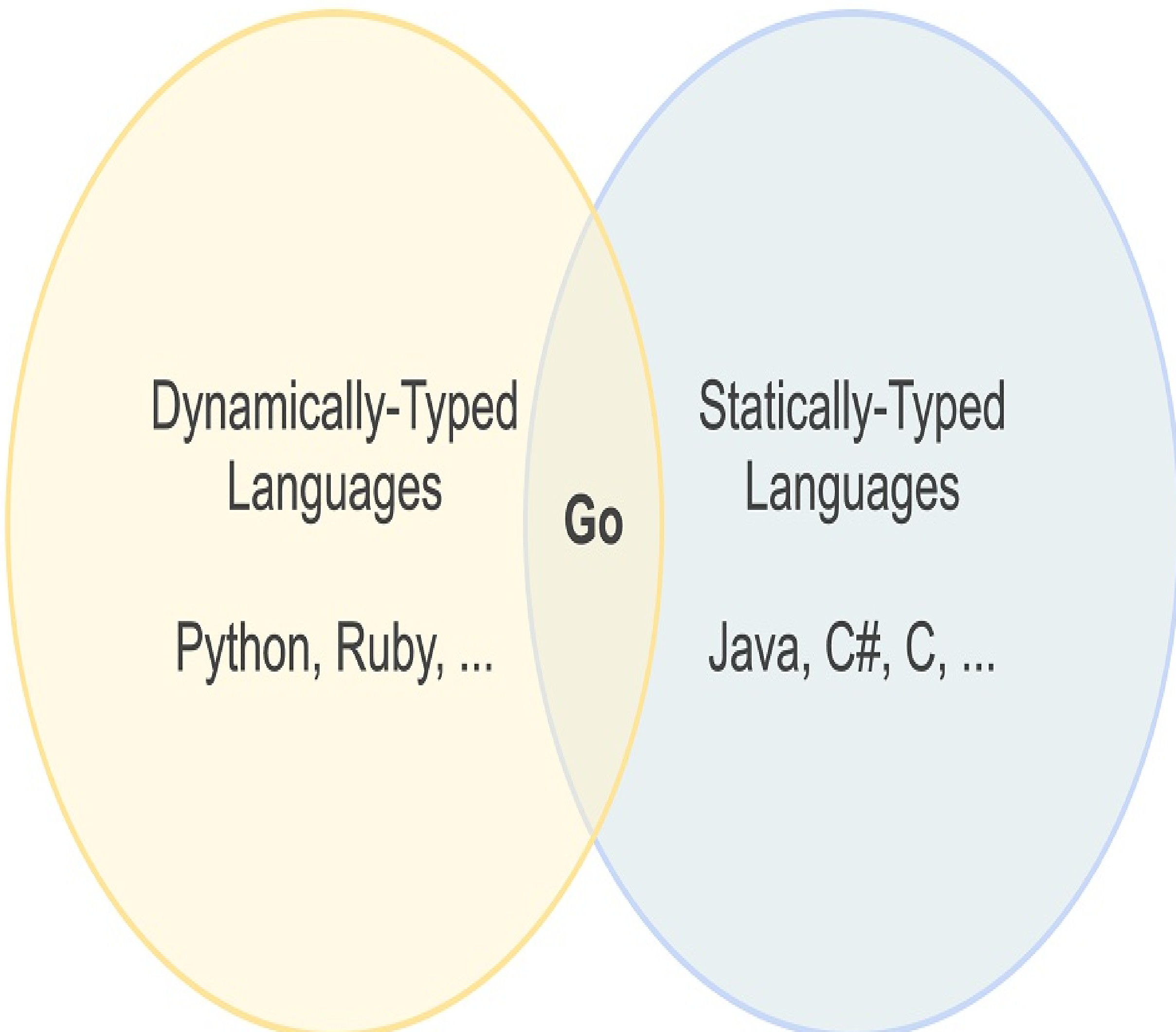
Of course, nothing comes for free. Here are some possible downsides of testing:

- *More code*—Tests are additional code you have to maintain.
- *More work*—Initially, you will have to add tests instead of what your customer wants. But in time, you might actually end up becoming faster. Otherwise, you would be manually testing and debugging your code.
- *Testing becomes the primary goal*—Our goal should be about designing a system consisting of loosely coupled, cohesive, and composable parts. Tests can definitely help you, but they are not your end goal. Testing should be pragmatic, and we should not test just for the sake of testing. If you take it too far, you might be going to the extreme end of the testing spectrum. There is a thin line there.

1.2 A brief tour of Go

Go is in a sweet spot between a dynamically-typed language like Python and a statically-typed language like C. You might have researched Go before writing code with it. While doing so, it is easy to miss critical knowledge that makes Go shine. We call a Go developer a Gopher, and I want you to be a sound Gopher. To do that, you need to know the roots of the language, the motivations, and its notable features from a bird's-eye view.

Figure 1.1 Go is in a sweet spot between a dynamically-typed and statically-typed programming language.



At the beginning of this section, you will learn about the motivations that led to the creation of Go. I think it would be hard to understand the design choices behind Go without knowing the background. After that, you will explore the differentiating and notable features of the Go language. You will see me comparing it to the other languages and where it stands among them. You will learn about what makes Go different. This background knowledge will shed light on what you will be learning throughout the book.

1.2.1 Motivations

It took five years of hard work to create the first stable release of Go. Back in 2007, three seasoned programmers from Google: Robert Griesemer, Rob Pike, and Ken Thompson, were sick of dealing with slow compilations, complex language features, and hard-to-understand code. Languages like C, C++, and Java were usually fast, but they were not developer-friendly. Other languages like Python, PHP, Ruby, and Javascript were developer-friendly but they were not efficient. This loss of motivation drove them to consider creating a new language that could solve all these problems, maybe more. They asked: "What should a modern, practical programming language look like?"

After working on the new language for two years, they released it as an open-source language to the public in November 2009. Over the three years, many people have contributed to the language with their ideas and code. Finally, they released Go 1.0 in 2012. You might ask: Why did it take so long? Go creators wanted to create a language by meticulously experimenting, mixing, and distilling the best ideas from many other languages. C, Modula, Newsqueak, Oberon, Pascal, and Smalltalk had a significant influence on the language's initial design:

- C-like statement and expression syntax.
- Pascal-like declaration syntax.
- Oberon-like packaging system. Instead of using public, private, and protected keywords to manage access to an identifier, Go and Oberon use a simple mechanism to export an identifier from a package. Oberon, like Go, when you import a package, you need to qualify the package's name to access the exported identifiers. Go exports when you capitalize the first letter, and Oberon does so when adding an asterisk.
- Smalltalk-like object-oriented programming style. Developers from

other object-oriented programming languages to Go are often surprised when they can't see any `classes`. There is no concept of `class`: `Data` and `behavior` are two distinct concepts in Go.
- Smalltalk-like `duck-typing` style in which you can pass a value to any type that expects a set of behaviors. You can see the same feature in other popular languages like Ruby and Python. But what makes Go different in this case is that Go provides type-safety and duck-typing at the same time.
- Newsqueak-like concurrency features. Newsqueak was another language created by Rob Pike.
- An object file format from Modula.

Their hard work paid off, and Go became popular by enabling developers to write simple, reliable, and efficient software. Today, millions of developers and many companies around the world are using Go to build software. Some notable companies using Go in production are Amazon, Apple, Adobe, AT&T, Disney, Docker, Dropbox, Google, Microsoft, Lyft, and so on.

Go vs. golang?

Rob Pike has proposed the language's name within an email that he sent on Sep 25th, 2007:

Subject: Re: prog lang discussion
From: Rob 'Commander' Pike
Date: Tue, Sep 25, 2007 at 3:12 PM
To: Robert Griesemer, Ken Thompson

i had a couple of thoughts on the drive home.

1. name

'go'. you can invent reasons for this name but it has nice properties.
it's short, easy to type. tools: goc, gol, goa. if there's an interactive
debugger/interpreter it could just be called 'go'. the suffix is .go
...

Disney had already registered the domain "go.com". So the Go team needed to register the domain golang.com instead and the word "golang" stuck with the language. To this day, most people call the language "golang". Of course, the actual name is Go, not golang. But in a practical sense, the golang keyword makes it easier to find something related to Go on the web. You can read more about the history of Go at this link: https://commandcenter.blogspot.com/2017/09/go-ten-years-and-climbing.html

1.2.2 What can you do with Go?

You might know that you can create robust web servers and cross-platform command-line tools if you know a little bit about Go. Then again, it wouldn't hurt to list some other sorts of programs that you can successfully develop with Go. Here are some of them:

- *Web services*—Go has a built-in `http` package for writing web servers, web clients, microservices, and serverless applications without installing third-party packages. It also has a database abstraction package called `database/sql` that you can use within your web application.
- *Cross-platform CLI tools*—As I've said above, you can create

interactive, fast, and reliable command-line tools. The best part is that you compile and run your program to work natively on a Linux distro, Windows, OS X, etc. For example, a bash-like shell, static website generators, compilers, interpreters, network tools, etc.

- *Distributed network programs*—Go has a built-in net package for building concurrent servers and clients, such as NATS, raft, etcd.
- *Databases*—People wrote many modern database software using Go, including Cockroach, Influxdb, GoLevelDB, and Prometheus.

Although it's possible, there are some areas where Go is not best for:

- *Games*—Go doesn't have built-in support for game development, but many libraries can help you build a game. For example, Ebiten and Pixel. However, it doesn't mean that you can't develop a Massively multiplayer online game server!
- *Desktop Programs*—Like game development, Go doesn't have built-in support for developing desktop applications. Some cross-platform packages can help Fyne and Wails.
- *Embedded Programs*—You can create embedded programs using third-party libraries: Gobot, TinyGo (*a Go compiler for low-resource systems*), and EMBD. Go also has a built-in package called *cgo* that allows you to call C-code (*or Go code from C*) within your program.

1.2.3 The reasons behind Go's success

Opinionated

There is often one right way to do things in Go. There are no tabs vs. spaces arguments in Go. It formats code in a standard style. Refuses to compile when there are unused variables and packages. Encourages packages to be simple and coherent units that mimic the Unix way of building software. Refuses to compile when there is a cyclic dependency between packages. Its type system is strict and does not allow inheritance, and the list goes on.

Simplicity

The language is easy to work with, concise, explicit, and easy to read and understand. It's minimal and easy to learn in a week or so. There is a 50-page long specification that defines the mechanics of the Go language. Whenever confusion about some language feature occurs, you can get an authoritative answer from the spec. The backward compatibility of Go guarantees that even though Go evolves each day, the code you wrote ten years ago still works today.

Type system and concurrency

Go is a strongly and statically typed programming language and takes the best tenets of object-oriented programming like composition. The compiler knows every value type and warns you if you make a mistake. Go is a modern language with built-in support for concurrency, and it's ready to tackle today's large-scale distributed applications.

Built-in packages and tools

Maybe newcomers believe that the Go Standard Library—*stdlib*—lacks features and depends on third-party code. But in reality, Go comes with a rich set of packages. For example, there are packages for writing command-line tools, http servers/clients, network programs, JSON encoders/decoders, file management, etc. Once newcomers have had enough experience in Go, most get rid of the third-party packages and prefer using the Standard Library instead.

When you install Go, it comes with many built-in tools that help you develop Go programs effectively: A compiler, tester, package manager, code formatters, static code analyzers, linters, test coverage, documentation, refactoring, performance optimization tools, and more.

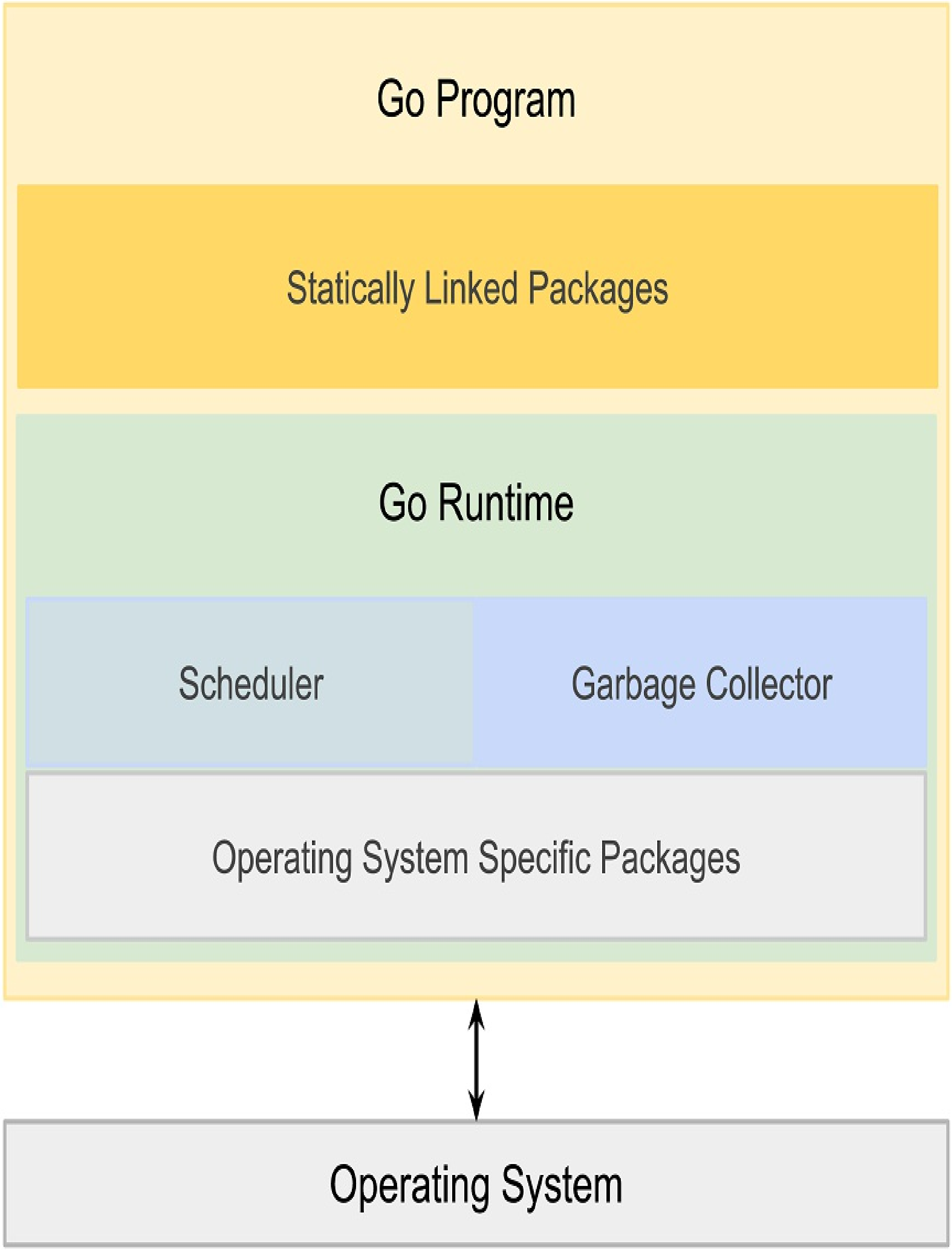
Go compiler

There is no intermediary execution environment such as an interpreter or a virtual machine. Go compiles code directly to fast native machine code. It's also cross-platform: You can compile and run your code on major operating systems like OS X, Linux, Windows, and more.

One of the design goals of Go from the beginning of its design was a fast compilation. Go compiles so fast that you may think you're not even compiling your code. It feels as if you're working in an interpreted language like Python. A fast compiler makes you productive by quickly writing and testing your code. So how does it compile so fast:

- The language grammar is simple and easier to parse.
- Each source code file tells the compiler what the code should import at the top. So the compiler does not need to parse the rest of the file to find out what the file is importing.
- The compilation ends if a source code file does not use an imported package.
- The compiler runs faster because there are no cyclic dependencies. For example, if package A imports package B, package B cannot import package A.
- During compilation, the compiler records the dependencies of a package and the package's dependencies on an object file. Then the compiler uses the object file as a caching mechanism and compiles progressively faster for subsequent packages.

Figure 1.2 Go Runtime layers. The Go compiler embeds the Go Runtime and operating-system-specific interfaces in an executable Go file.


Go Program
Statically Linked Packages
Go Runtime
Scheduler
Garbage Collector
Operating System Specific Packages
Operating System

In Go, compiled code generates a single executable file without external dependencies; everything is built-in, as you see in figure 1.2. Every Go program comes with an integrated runtime that includes:

- A garbage collector that runs in the background and automatically frees up unused computer memory. For example, in C and C++, you need to manage memory manually.
- A goroutine scheduler that manages lightweight user-space threads called goroutines. The garbage collector also uses the goroutine mechanism.

Since the executable file doesn't have any dependencies, deploying your code to production becomes trivial. On the other hand, if you were to run Java code on production, you would need to install a virtual machine. With Python, you would need to install a Python interpreter. Fortunately, since Go compiles into machine code, it works without an interpreter or virtual machine.

Note

You can read more about the design mechanics behind the Go compiler at the link: https://talks.golang.org/2012/splash.article#TOC_7.

1.2.4 Type system

"If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se but rather implementation inheritance. Interface inheritance is preferable. You should avoid implementation inheritance whenever possible.

— James Gosling (Java's inventor)

The Go type system is straightforward and orthogonal. Orthogonal in the sense that each feature is independent and can be combined freely in creative ways. Orthogonality and simplicity allow us to use the type system creatively that even Go creators might not have imagined.

Object-Oriented but without classes

Go is an object-oriented language, but there are no classes and inheritance. Instead of classes, every type can have behaviors. Rather than inheritance, gophers create polymorphic types using interfaces. Instead of building big classes, you *compose* bigger things from small things. If some of these things are not clear yet, don't worry. You're going to learn some of them here.

The first object-oriented Programming language was Simula, and it introduced classes, objects, and inheritance. Then SmallTalk came around, and it was about message passing. Go follows the SmallTalk tradition of message passing to some extent.

The most general concept in modern object-oriented programming languages is *class*, where you put behavior and data together. By data, I mean the variables in which you store data, and by behavior, I'm talking about functions and methods that can transform data from one shape to another. For example, Java is one of those languages that embrace the class concept as a core language feature. Java classes combine data and behavior and disallow you to define anything outside of a class. Ruby and Python are more relaxed in this sense and define behavior out of a class definition.

In the following Java code example, the *data*—`host` variable—and the *behavior*—the `Start` method—are tightly attached to the `Service` class.

```
public class Service {
    private String host;                      // data
    public void Start() throws IOException { // behavior
        ...
    }
}
```

As you've seen, Go mixes and distills the best features of many languages that came before it. Go is an object-oriented-Programming language but not quite so in the mainstream sense. You can write a similar type in Go as follows:

```
type Service struct {
    host string    // data
}
```

As you can see, `Service` is a *struct* and similar to a *class* but consists only of data. On the other hand, the behavior is entirely separate from the data as follows:

```
func Start(s *Service) error { // behavior
    ...
}
```

There is nothing that tightens you to a class where you define behavior and data together. In this sense, I can say that Go is closer to procedural languages than mainstream object-oriented programming languages. This separation allows you to mix data with behavior without thinking about designing so-called classes from day one. Unlike most other object-oriented programming languages, there is no hierarchy of classes.

Every concrete type can have behaviors

The Go methods are simple functions. For example, if you want to *attach* the `Start` function above as a method to the `Service` type, you can do it easily like so:

```
func (s *Service) Start() error {
    // s is a variable in this scope
    fmt.Println("connecting to", s.host)

    return nil // no errors occurred
}
```

As you used to from other OOP languages, the `Service` type has a `Start` method now. But, there is no "`this`" keyword in Go. Instead, each method has a receiver variable. In the example above, the receiver is the variable `s` and its type is `*Service`. The `Start` method can use the `s` variable and reach out to the `host` field.

Let's create a new `Service` value:

```
svc := &Service{host: "localhost"}
As you can see, there are no constructors in Go. The code above d
err := svc.Start()
if err != nil {
    // handle the error: log or fail
```

```
}
```

You will see the following after calling the `Start` method:

```
Connecting to localhost
```

Let's create one more service value:

```
svc2 := &Service{}       // makes an empty *Service value
svc2.host = "google.com" // assigns "google.com" to the host fiel
svc2.Start()
```

This time, you'll see the following:

```
Connecting to google.com
```

As you can see, each `*Service` value is like an instance of a class.

Go is pass-by-value. If you hadn't added a pointer receiver (`*Service`), you wouldn't be able to change the `host` field. It's because each time you call the method, Go would copy the receiver variable and you would be changing the host field of another `Service` value.

Behind the scenes, the compiler registers the `Start` method to a hidden list called the *method set* of the `*Service` type. So that the compiler can call the method as follows:

```
err := (*Service).Start(svc)
// Connecting to localhost
```

In the example above, you call the `Start` method on the `svc` value. You can also call it on the other value:

```
err := (*Service).Start(svc2)
// Connecting to google.com
```

Since the `Start` method is in the `Service` type, the compiler needs to go through the type first and call the method. But you don't have to do that. The best approach is calling the `Start` method on a value as follows:

```
svc.Start()
svc2.Start()
```

Another difference of Go is that you can *attach behaviors to any concrete type* you have, whether you defined the type long before as long as it is in the same package. This design allows you to *enrich* types with behavior whenever you want:

```
type number int
func (n number) square() number {
    return n * n
}
```

Since behavior and data are separate things, if you want to add another method to the `number` type, you don't need to change the definition of the `number` type. This feature allows you to evolve your code without changing the existing code:

```
func (n number) cube() number {
    return n.square() * n
}
```

Now you can call all the methods on a value of the `number` type:

```
var n number = 5
n = n.square()        // same as: square(n) and returns 25.
n = n.cube()          // returns 15625.
```

Interfaces unlock polymorphic behavior

As you can see, Go provides a different kind of object-oriented programming where data and behavior are two different things. While classes and objects are essential in other object-oriented programming languages, it is the behaviors that matter in Go. As an example, let's take a look at the `Writer` interface of the `io` package:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

The `Writer` interface only describes the behavior without an implementation. It has a single method that only describes writing to something with a `Write` method. Any type with a `Write` method can be a `Writer`. When a type implements all the methods of an interface, we say that type satisfies the interface. For example, the `File` type is a `Writer`, and we say it satisfies the `Writer` interface. The `Buffer` type is also a `Writer`. They both implement the same `Write` method of the `Writer` interface:

```
type File struct { ... }
func (f *File) Write(b []byte) (n int, err error) {
    ...
}
type Buffer struct { ... }
func (b *Buffer) Write(p []byte) (n int, err error) { ... }
```

Suppose that you want to write a text message to a file. To do that, you can use a function called `Fprint` that takes the `io.Writer` interface as argument and writes the message to any given value of a type that has a `Write` method:

```
func Fprint(w io.Writer, ...)
```

First, you're going to open the file using the stdlib's `os` package, and then you will pass it to the function:

```
f, _ := os.OpenFile("error.log", ...)
Fprint(f, "out of memory")
```

Let's say, instead of writing the error message to a file, let's write it to an in-memory buffer using the stdlib's `Buffer` type:

```
var b bytes.Buffer
Fprint(b, "out of memory")
```

The `File` and `Buffer` types say nothing about the `Writer` interface. All they do is implement the `Write` method of the `Writer` interface. Beyond that, they don't know anything about the `Writer` interface. Go does not couple types to interfaces, and types implicitly satisfy interfaces. You don't even need the `Writer` interface, and you could still describe the behavior without it.

In contrast to the Go interfaces, Java doesn't allow implicit interfaces and types to become coupled to interfaces. Let's take a look at classical Java interface and classes that implement a similar interface:

```
public interface Writer {
```

```
    int Write(p byte[]) throws Exception
}
// File couples itself to the Writer interface.
public class File implements Writer {
    ...
}
```

In the above example, the `File` type needed to denote that it implements the `Writer` interface. By doing so, it couples itself to the `Writer` interface, and coupling is terrible in terms of maintainability. Suppose many types implement the interface, and you add another method to the interface. In that case, you would also need to add the new method to every type that implements the interface.

Another problem can arise if the `File` type doesn't explicitly say that it implements the `Writer` interface. Suppose that there is a function that takes the `Writer` interface as an argument. You wouldn't be able to pass it as a `File` object even though the `File` has a `Write` method.

In Go, there isn't a problem like that. The interface name does not matter. You can pass a value to a function that takes the `Writer` interface as long as the type of the value has a `Write` method with the same signature (with the same input and result values).

Note

Go will complain at compile-time if you pass a value that does not satisfy an interface.

For example, let's declare a new interface as follows:

```
type writer interface {
    Write(p []byte) (n int, err error)
}
```

Then, let's declare a function as follows that takes the `writer` interface:

```
func write(w writer, ...)
write(b, "out of memory")
write(f, "out of memory")
```

You can pass a `*File` or `*Buffer` value to the `write` function because each has a `Write` method with the same signature declared in the `writer` interface. The function could still take the same values even if you had declared it as follows:

```
func write(w io.Writer, ...)
write(b, "out of memory")
write(f, "out of memory")
```

As I said, the interface names do not matter. Only the method signatures should match.

Inheritance vs. Embedding

"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got were a gorilla holding the banana and the entire jungle."

— Joe Armstrong

In classical object-oriented programming languages, a *child class* can reuse functionality and data by *inheriting* from the *parent* or *base class*. The problem with inheritance is that the child and parent classes become coupled. Whenever one of them changes, the other one usually follows and leads to a maintainability nightmare. There are many other problems with inheritance, but I won't discuss them but focus on how Go approaches code reusability.

Go does not support inheritance in the classical senses but supports composition instead. You can reuse functionality and data from other types using a feature called *struct embedding*. Let's say there is a type that can store information about a file:

```
type resource struct {
    path string
    data []byte
}
```

Let's add a method that denies access to the resource type:

```
// deny denies access to the resource.
```

```
func (r *resource) deny() {
    ...
}
```

Now, you can use it like so:

```
errLog := &resource{path: "error.log"}
errLog.deny()
```

Now you want to store additional information about an image file, but you don't want to duplicate the data and behavior of the resource type. Instead, you can *embed* the `resource` type in the image type:

```
type image struct {
    r resource
    format imageFormat // png, jpg, ...
}
```

Finally, you can use the image type as follows:

```
img := &image{
    resource{path: "gopher.png"},
    format: PNG,
}
```

Figure 1.3 Inheritance vs. Composition. On the left, the image type inherits from the resource type and creates a type hierarchy. On the right, the image type embeds a value of the resource type.

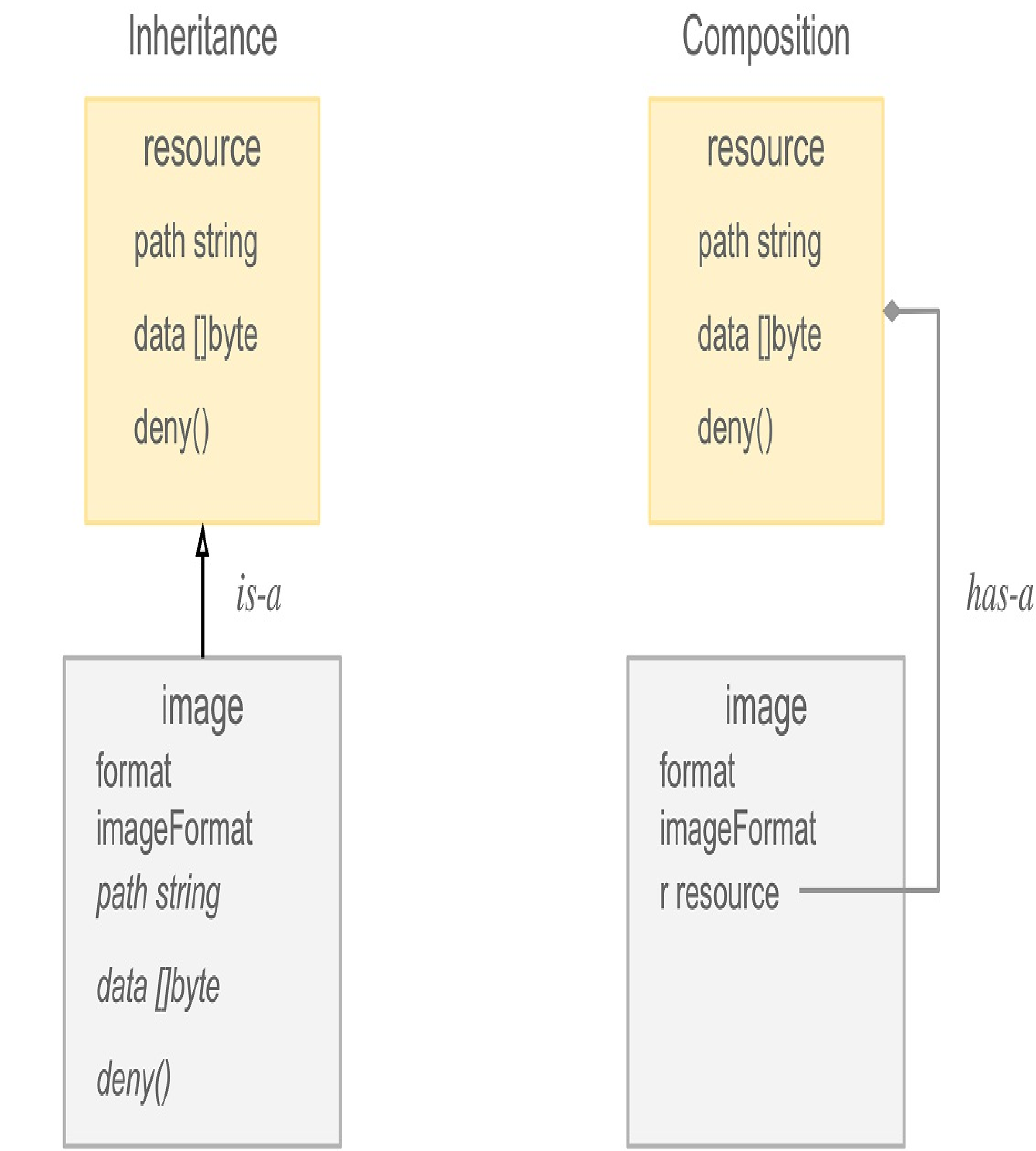


image *is-a* resource type
image *copies* data and behavior
from resource onto itself

resource *is a part-of* image
image *has* a resource value
but it's *not* a resource

Most of the other object-oriented languages call it inheritance and copy the *path* and *data fields*, as well as the deny *method* to the image type, and the following code would work:

```
img.path
img.deny()
```

Whereas Go only embeds a resource value into the image value. So the path and data fields and the deny method still belong to the resource type:

```
img.format   // PNG
img.r.path   // "gopher.png"
img.r.deny() // calls the deny method of the embedded resource
```

As you can see, you're using the embedded resource value as an image value field. This technique is called composition, in which the image type has a resource value. The image value will always have a resource value, and you can even change it on the fly while running your program. Although Go uses embedding, you can also mimic inheritance and make the image pretend to be inheriting from the resource type:

```
type image struct {
    resource
    format imageFormat // png, jpg, ...
}
```

Have you noticed that the embedded resource value doesn't have a field name? So now you can use it as follows:

```
img.path   // same as: img.resource.path
img.deny() // same as: img.resource.deny()
```

When you don't directly use the embedded `resource` value and type: `img.path`, behind the scenes, the compiler types: `img.resource.path`. Likewise, when you call the `deny` method on the image value, the compiler forwards the call to the `deny` method of the embedded resource: `img.resource.deny()`. But the `path` field and the `deny` method still belong to the `resource`, not the `image`.

In other object-oriented languages, there is an is-a relationship between a parent and child class. So you can use the child type where a parent type is expected. For example, let's say you want to deny access to all resources. Let's try putting them in a slice using the common resource type and then deny access to each in a loop:

```
// assume the video type embeds the resource type
vid := &video{resource: {...}}
resources := []resource{img, vid}
for _, r := range resources {
    r.deny()
}
```

But you can't. The `image`, `video`, and `resource` are different types. If they were similar types, you could put them in the `resources` slice. The `image` type would be a `resource` type if you were to inherit it from the `resource` type. Since there is no inheritance in Go, you used composition. So how can you put these different types of values in the same slice? You need to think of a solution from the way Go approaches object-oriented programming: They share a common behavior called `deny`, and you can represent that behavior through an interface:

```
type denier interface {
    deny() error
}
```

Now you can put different types of values in the same slice as long as each one implements the `deny` method. Then you can deny access to every resource at once using a loop:

```
for _, r := range []denier{img, vid} {
    r.deny()
}
```

The code above will work. In Go, you achieve the *is-a* relationship between types using interface polymorphism. In contrast to other object-oriented languages, you group types by behavior rather than data. You use interfaces for polymorphic behavior and embedding for some degree of reusability. Both of these features allow us to design loosely coupled components without creating fragile type hierarchies.

1.2.5 Concurrency

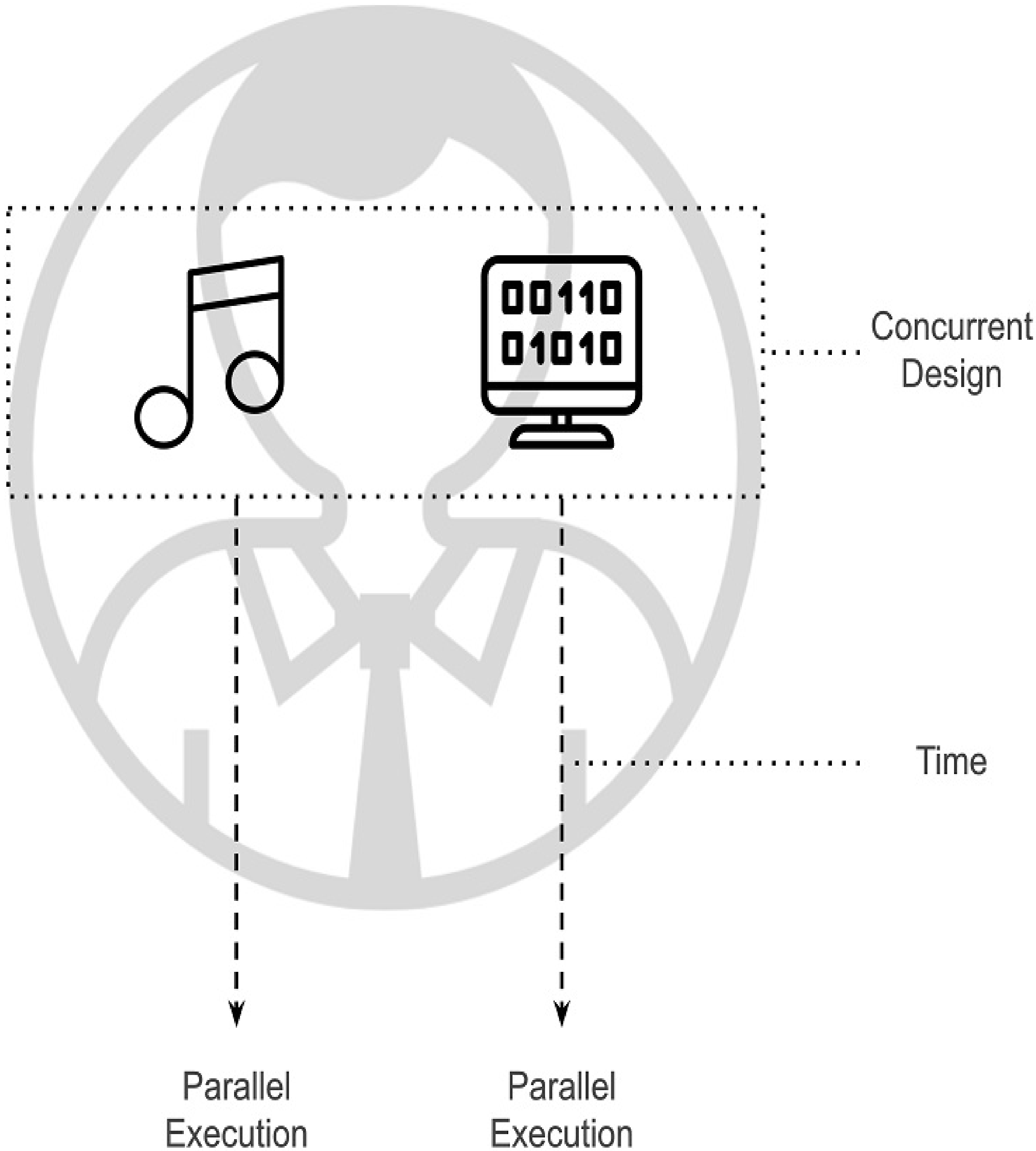
In the last section, you learned about the roots of the Go language and what sort of programs you can build with it. You also learned about some of the things that make Go different from other languages. In this section, you will expand your horizon and learn about some other notable features of Go that separate it from other languages. There are many things that I want to explain, but it is impossible to cover all of them in the same chapter.

Concurrency vs. Parallelism

Go is a concurrent programming language that provides a way of abstracting and managing concurrent work. Newcomers often use concurrency as a way to speed up their programs, but that's only half of the story. The goal of concurrency is not about creating fast-performing programs, but it could be a by-product. Instead, it's about how you design your program before you run it. Parallelism is about performance, which may happen only after running your program.

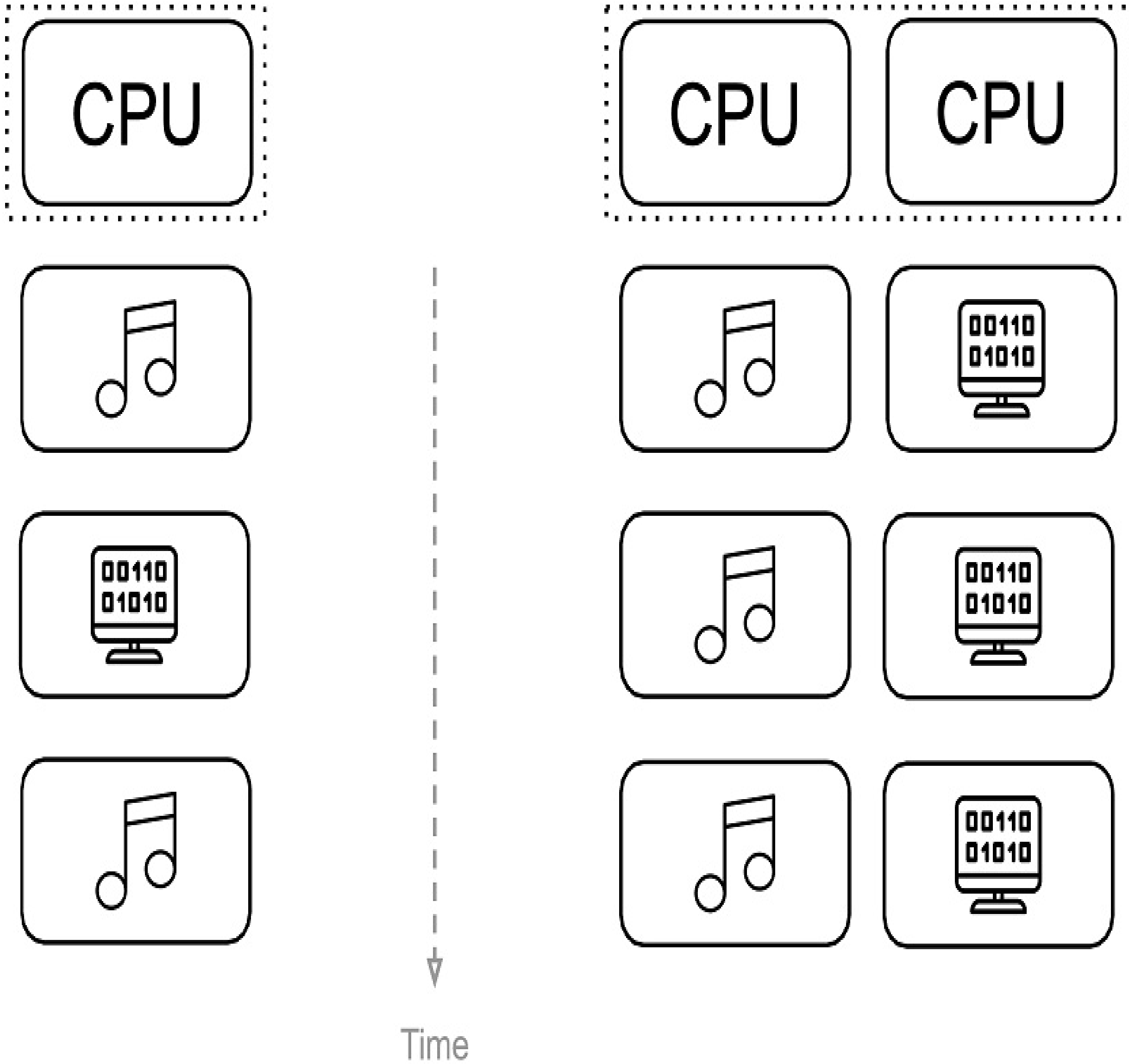
Let's imagine a scenario from our daily lives. You are listening to music while writing code in your favorite editor. Is what you're doing parallel or concurrent? Apart from a philosophical and scientific point of view, in a practical sense, what you do is parallel because you are coding while enjoying the music at the same time. Even if you are not alive, your senses of hearing, touch, and vision would still have a concurrent structure. But if you were dead, your concurrently structured senses would no longer be able to do parallel processing.

Figure 1.4 A programmer has a concurrent physical structure to listen to music and write code in parallel.



Let's look at the situation through the lenses of your operating system. Assuming you only have a single processing unit on your machine, the operating system will quickly share it between the media player and coding editor, and you won't even notice it. It will allow the processor to run the media player for a while, then stop the player and instruct the processor to run the coding editor for a while. This work can only become parallel when multiple processing units are available to the operating system.

Figure 1.5 On the left, a single processing unit (CPU) is running one task at a time. On the right, multiple CPUs are simultaneously running a media player and code editor at the same time (in parallel).



Concurrency features

Dividing concurrent work into independent processing units allows you to design comprehensible and neat concurrent code as if you were writing sequential code.

Tony Hoare invented a concurrency model called Communicating Sequential Processes (CSP) in a 1978 paper in which anonymous processes sequentially execute statements and communicate by sending messages to each other[1]. The basic principles of the Go concurrency model come from CSP:

- *Goroutines*—Lightweight user-space threads with minimal overhead that you can program for like a normal function.
- *Channels*—Type-safe values for communicating between goroutines in a concurrency-safe manner without sharing memory, threads, and mutexes.
- *Select statements*—A mechanism for managing multiple channel operations.
- *Scheduler*—Automatically manages and multiplexes goroutines over operating system threads. There are still threads, but they're hidden from your view and only visible to the *Go scheduler by default*.

Sometimes you also need to use classical concurrency features such as a mutex, and Go supports them, but instead of using them, we try to use channels. In this chapter, you will only be learning about goroutines and channels.

Goroutines vs. threads

An operating system process is also divided into smaller units called threads. For example, a media player program is a process. It may have several threads: One for streaming music data from the Internet, one to process the data, and yet another for sending the data to a couple of speakers on your computer.

The operating system has a mechanism to switch between these threads for running them concurrently, usually in the kernel. Switching between these threads can be costly due to the overhead of making system calls while making *context switches*. When a context switch occurs, the operating system saves the current thread state to memory and then restores the previous state of the next thread. That's a costly operation, and that's why we use goroutines while programming with Go.

Go scheduler is Go's way of managing goroutines. Since goroutines are cheap, the scheduler can run millions of them on a small set of kernel threads. The scheduler is also vital for efficient communication between goroutines via channels. And it also manages other runtime goroutines like the garbage collector. You will learn about channels soon.

Note

You might want to watch this video that explains more details about the Go scheduler: https://www.youtube.com/watch?v=YHRO5WQGh0k.

Goroutines

Every Go program has a function called main as an entry point to the program. And, when you execute a program, the operating system runs the main function on a goroutine called the main goroutine:

```
func main() {
    ... the main goroutine runs the code here …
}
```

Figure 1.6 The main Goroutine is running the main function code. The arrow depicts the passage of time.

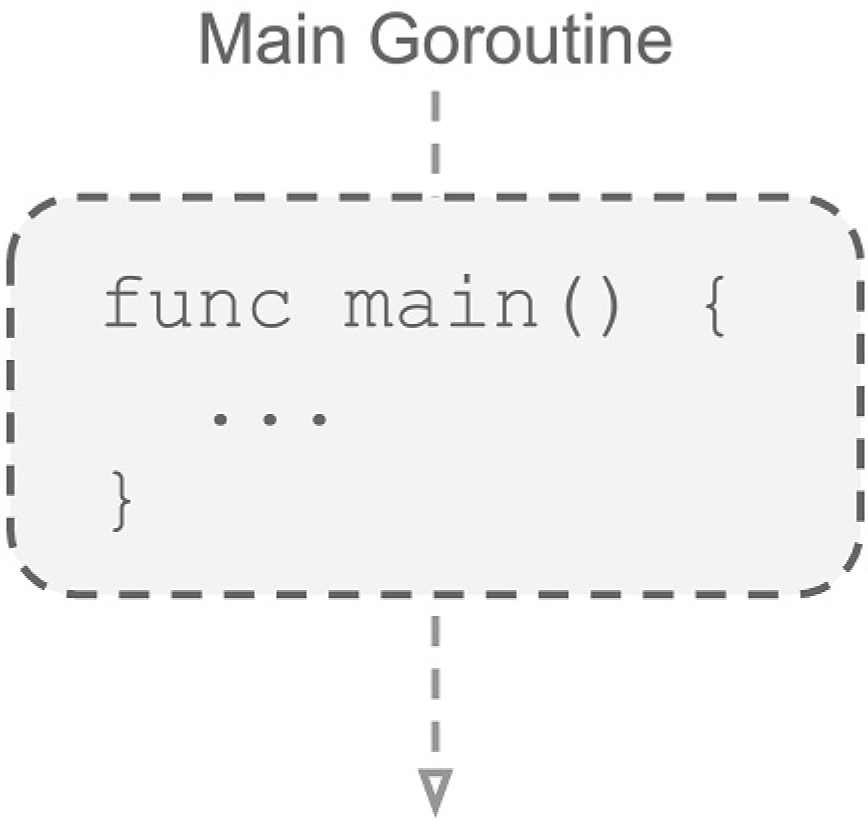
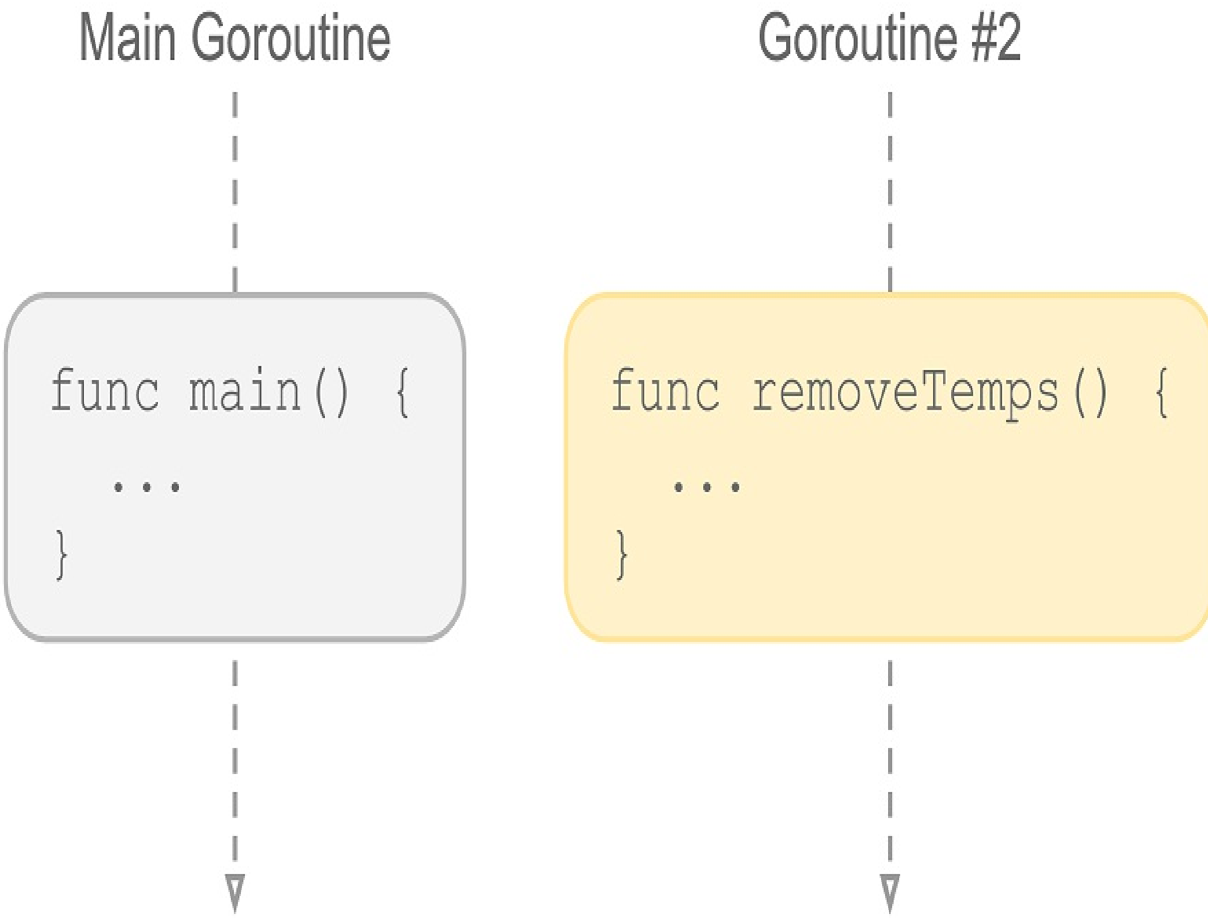


Figure 1.6 shows that every program has at least a single goroutine called the main goroutine. The main goroutine runs the main function. The program along with the main goroutine terminates when the main function ends.

Of course, Go programs can have millions of goroutines. Say you're writing a database server that creates many temporary files. You can write a function using a simple `go` statement and then run it in the background as a goroutine. The rest of the code will continue executing without waiting for the goroutine to finish:

```
go removeTemps()
// ... rest of the code ...
```

Figure 1.7 Two Goroutines are running concurrently. They can only run in parallel if there are multiple processing units.



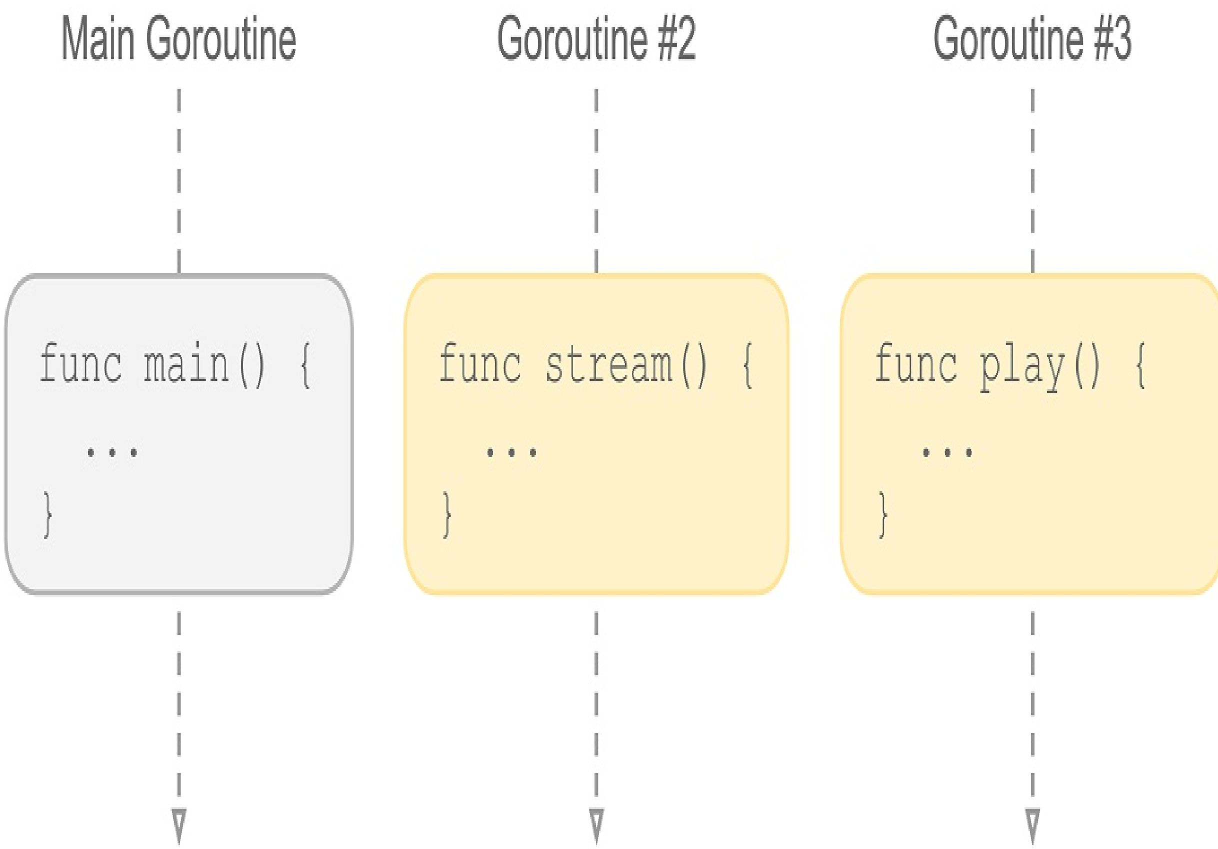
In Figure 1.7:

1. The main function starts running in the main goroutine.
2. Then the main function launches a new goroutine called `removeTemps`.
3. Both goroutines run concurrently. Or, in parallel, if there are multiple process units.

Think about the media player example I previously showed you. You may design it with two goroutines: While one goroutine is streaming a song, the other will play it. These two goroutines may work concurrently while the other code in the program keeps working:

```
go stream()
go play()
... the main goroutine keeps running the rest of the code …
```

Figure 1.8 Three Goroutines are running concurrently.



In Figure 1.8:

1. The main goroutine launches the program by executing the main function.
2. Then the main function launches two goroutines: `stream` and `play`.
3. These three goroutines will keep running concurrently until the program ends.

Sometimes, you launch a goroutine but forget to end it. It's called a goroutine leak that unnecessarily keeps consuming system resources. That's why you should plan for how to exit them before launching goroutines.

Channels

You need to find a way to provide data to the play goroutine from the streaming goroutine for playing a song. Since they work concurrently, you cannot just go and share data between them. Otherwise, you would have to deal with corrupted data. Fortunately, there is another synchronization mechanism that you can use: *channels*. Every channel has a data type that it can transmit. For now, you can think of channels as *data cables* or Unix pipes between goroutines:

```
cable := make(chan string)
```

Notice that channels are values like every other value. You created a channel variable in the code above and assigned it a channel that can only transmit the `string` data type values. In a real-world program, you would use a specific data structure. A channel value is just a pointer to a data structure so that you can pass the same channel value to functions as an argument. Suppose that the stream function receives a piece of data from the Internet. So you can send it to the channel that you created above using a *send statement*:

```
func stream(out chan string) {
    for {
        out <- "...fetched data..."
        ...
    }
}
```

The `stream` function takes a single input value called `out`, and its type is a string channel (`chan string`). These two-letter symbols "`<-`" are called the *receive operator*, or a *send statement* depending on where you put them around a channel value, but I'll call them *arrows* for now. If the arrow points to a channel as above, you *send* a value to the channel. In the other case, it means you *receive* a value from the channel:

```
func play(in chan string) {
    for {
        data := <-in
        ...
    }
}
```

When the stream goroutine sends a value to the channel, it will temporarily stop working until the play goroutine comes up and takes the value. On the other hand, if the play goroutine tries to receive from the channel, the same thing happens. There are also buffered channels where these constraints are more relaxed, but you will learn about them later in the book.

1.3 Summary

What you've seen so far is only the tip of the iceberg. Then again, I hope you understand that Go is a simple but powerful modern language that hides complexity behind easy-to-use language features.

- Well-designed code in Go demands a different mindset, and well-designed programs are easy to maintain, reliable, and easy to test.
- Testing increases your confidence in your code and may make it less buggy.
- Go feels like a dynamic language, but at the same time, type-safe.
- The type system gets the best parts of object-oriented programming and promotes composition instead of inheritance.
- Concurrency is built-in into the language and provides a modern way of building concurrent programs.

[1] https://en.wikipedia.org/wiki/Communicating_sequential_processes

2 Getting Started with Testing

This chapter covers

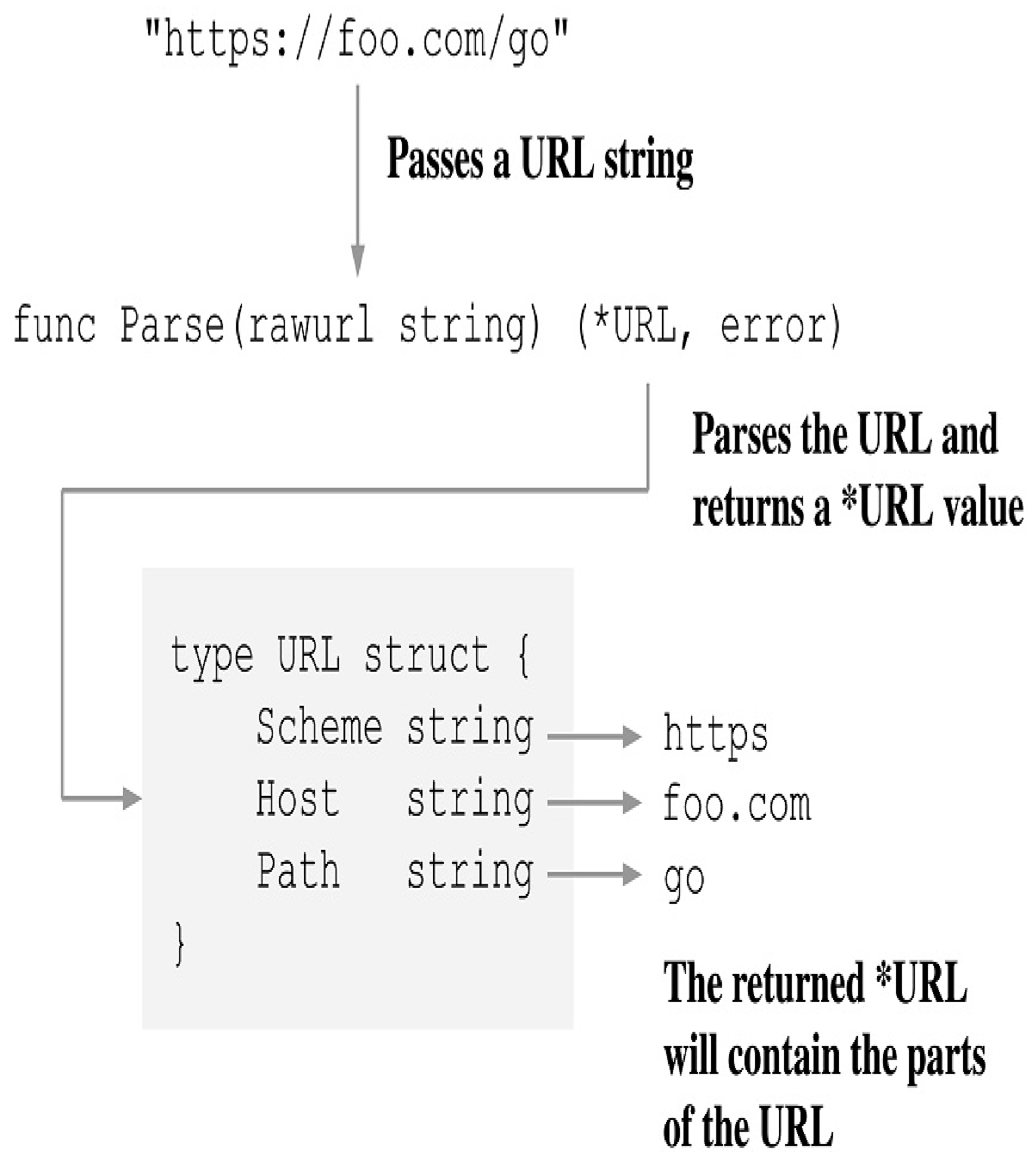
- Basics of unit testing
- Basics of Go testing framework
- Writing idiomatic tests and code in Go
- Adding a URL parser package to the Go Standard Library

Let’s go back in time. The Go team at Google is busy writing the Go Standard Library. You, the programmer, recently joined the team. At Google, other developers in a team called the Wizards are working on a redirection service.

When their program receives a web request, they check the URL to see if it's valid, and if the URL is valid, they want to change some parts of it and redirect the web request to a new location. But they realized that there isn't a URL parser in the Go Standard Library.

So they asked you to add such a URL parser package to the Go Standard Library. You know the basics but don't yet know how to write and test idiomatic code in Go. I'll work throughout the chapter to help you write idiomatic code by implementing and unit testing a new library package called *url* from scratch.

Figure 2.1 Parsing a URL



1. Parse a URL to check whether it's a valid URL
2. Separate the URL into its parts: Scheme, host, and path
3. Provide an ability to change the parts of a parsed URL

As you can see in Figure 2.1, the code passes a URL string to the `Parse` function. Then, `Parse` parses it, creates a new URL value, and returns a pointer to the URL value. It returns a pointer so that you can change the fields of the same URL value.

The `URL` struct type contains the URL parts, such as scheme, host, and path. More things are involved in a full-fledged parser, but let's keep things more manageable and only parse the scheme, hostname, and URL path.

First, you will learn how Go approaches testing, the definition of the unit in Go, and unit testing. After that, you will learn how to write your first unit test. After creating a basic test, you will learn how to communicate with the testing framework. You'll also learn how to write descriptive failure messages. After writing your first test, you will start writing and adding more tests to the url package.

You'll see what kind of errors you can get when you don't write a proper test function. So that you'll see the reasoning behind every line of code you'll be writing. Not only will you learn how to test, but you will also see some of the code behind the Go testing framework and understand it more deeply.

Alright, let's get started!

2.1 Go's testing approach

Let's start talking about how Go approaches testing before you start writing the url package. In Go, almost everything is built-in. This approach applies to testing as well. So you can automatically test your code using the built-in testing framework without installing any external tools or libraries. As with everything in Go, testing is also simple on the surface, but it hides the complexity behind simple programming interfaces.

Even though everything is built-in, there isn't a full-fledged testing framework per se other than a test runner and some testing packages. But, that doesn't mean that built-in testing facilities in Go are weak. The Go Standard Library is very powerful and provides a lot of helpers for testing.

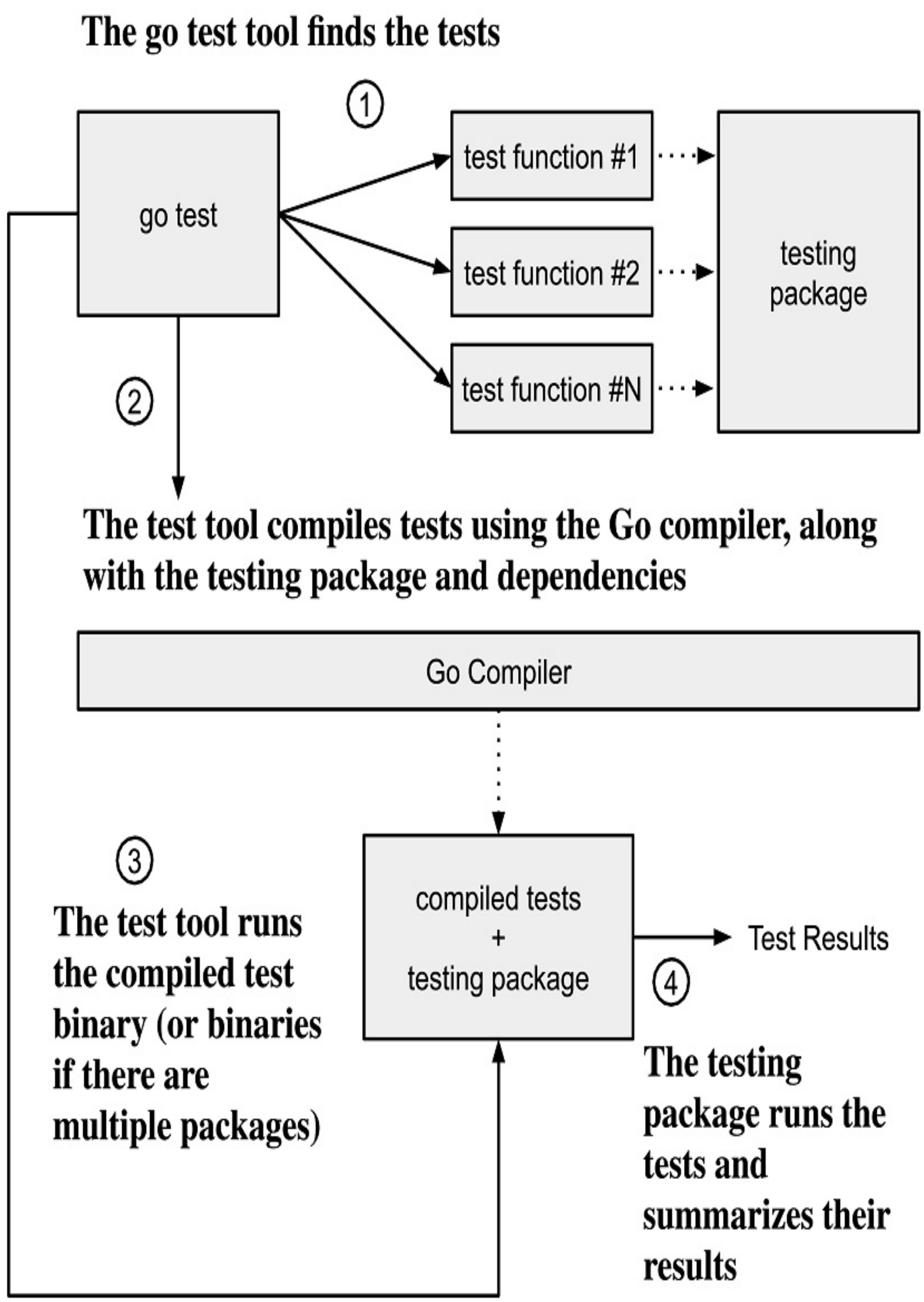
Still, many new Go developers immediately start looking for additional testing frameworks and packages before starting programming in Go. I understand them because I was not different. Coming from Node.js, I used to have some other third-party test frameworks for testing. Now, I mostly use the built-in testing facilities and bring little helpers when they are vital. Soon, you will find out that the Go testing tools have everything you need for almost every type of test.

For testing in Go, there are two main actors:

- *The test tool*—It's a simple command-line tool that comes built-in with Go. You can use the `go test` command to compile and run tests automatically. The go test tool finds tests, compiles them using the Go compiler, and runs the final test binary.
- *The testing package*—The test tool is only responsible for compiling and creating the necessary environment for the testing package. It is the testing package that helps you write and run tests. Then it reports a summary about whether your tests fail or succeed.

Let's take a look at Figure 2.2 to see the flow of the testing framework.

Figure 2.2 The Go testing framework mechanics


The go test tool finds the tests
1
go test
test function #1
test function #2
test function #N
testing
package
2
The test tool compiles tests using the Go compiler, along
with the testing package and dependencies
Go Compiler
3
The test tool runs
the compiled test
binary (or binaries
if there are
multiple packages)
compiled tests
+
testing package
Test Results
4
The testing
package runs the
tests and
summarizes their
results

1. First, the test tool finds the test functions and they communicate with the testing package, such as reporting whether they succeed or fail.
2. The test tool itself is not a compiler, so it uses the Go compiler to compile the test functions and the packages they depend on, and packs them in an executable binary.
3. The test tool runs the final test binary, and the testing package takes control.
4. The testing package runs the test functions in the executable binary and displays their results to the console.

In this chapter, you'll learn about the test tool and testing package, which I will occasionally refer to as the Go testing framework. There are other helper packages and tools in the Go Standard Library and tools as well. I will reveal them as you read the book.

2.1.1 What is a "unit" in Go?

Many newcomers often get confused, especially when defining the word "unit" in Go. So in this section, I will explain the meanings of "unit" and "unit testing" in the Go language. But all those terms are vague. People interpret them very differently everywhere. You can ask a group of people what a unit or unit test is, and you get many different answers.

In some other popular programming languages, classes are the primary actors, and developers write unit tests against classes. But there are no classes in Go; instead, you organize code around packages, and each package ideally provides something unique to other packages.

Often, a package can get large, and it may consist of many functions and types. When that happens, you can separate the package code to multiple files in the same package for more straightforward navigation. For example, the stdlib's strings package is relatively large, combining relevant functionality into different source-code and test files, as seen in Table 2.1.

Table 2.1. The strings package

Types and Functions	Code File	Test File

`Fields(s string) []string`
 	
A function that splits a string value by spaces.
 	
`HasPrefix(s, prefix string) bool`
 	
A function that checks whether a string begins with another string.
 	
There are dozens of other functions in the strings package as well.
	`strings/strings.go`
	
	This file declares the functions.
		`strings/strin`
		
		This file tests al functions that ha declared in the s file.

`Builder struct` A type that can efficiently combine strings into a buffer.	`strings/builder.go` This file declares the Builder type and its methods and some other helper functions.	`strings/build` This file contair for the code in t `builder.go`.
`Reader struct` A type that can read from a data stream.	`strings/reader.go` This file declares the `Reader` type and its methods.	`strings/reade` This file contair for the code in t `reader.go`.
`Replacer struct`	`strings/replace.go`	`strings/repla`

A type that can replace a series of strings with replacements.	This file declares the `Replacer` type and some other helper types and functions for `Replacer`.	This file contair for the code in t `replace.go`.

As you can see in table 2.1, all those functions, types, and tests together make a single package called *strings*. Separating the tests into multiple files still makes those tests the unit tests of the strings package because they only test the strings package.

Although the strings package is a unit in Go terminology, the types are also units, or perhaps, sub-units. For example, the `strings.Builder` type is also a unit and has its own unit tests.

As you can see, it is not always easy to define what is a unit or not. I believe the definition depends on your team. To me, a unit in Go is a *package*, and the description of a *unit test* is a test that verifies the behavior of a single package.

2.1.2 What is a unit test?

For example, a single unit test can verify if a function is working correctly. Let's say there is a simple function that adds two given numbers and returns the result:

```
func sum(a, b int) int {
    return a + b
}
```

Then you can write a unit test to verify if the sum function works correctly:

```
func main() {
    if n := sum(2, 3); n != 5 {
        fmt.Printf("TEST FAILED")
    }
}
```

Note

This example demonstrates a unit test but not an idiomatic way of unit testing in Go. This chapter will show you how to write idiomatic unit tests using the Go Standard Library's testing package.

Unit tests are the fundamental part of automated testing, and when crafted correctly, they are fast to run and help you design testable code. To me, a proper unit test often has the following characteristics:

- *Isolated*—A unit test usually verifies the logic of a small part of code in isolation.
- *Fast*—A unit test runs quickly and gives immediate feedback about the code.
- *Deterministic*—A unit test is consistent and gives the same result every time it runs. The things that you can't control are the leading causes of *flaky* (*indeterministic*) tests. And, a unit test should depend on other tests (except the test helpers) .

I think the last two characteristics are pretty much straightforward, but what about the first one: "Isolated"? So the real question is: What is a small part of code? What does it mean to be "small"? The scope of a unit test largely depends on a development team. The term doesn't matter as long as developers agree that a test is a unit test in their own problem space. If you ask me, a unit test is often a test that verifies single or multiple functions of an individual package and is written by developers.

2.1.3 Wrap up

Before moving on to the next section, let's summarize what you've learned so far:

- You can test your code automatically using the built-in testing framework without installing any external tools or libraries.
- The test tool uses the Go compiler to compile your tests and their dependencies and the testing package. Then it runs the compiled test binary.
- The testing package runs your tests and summarizes their results when the test tool runs the compiled test binary.

- Tests import and communicate with the testing package.
- Unit tests are usually fast, isolated, and deterministic.

2.2 Writing your first unit test

As I explained in the chapter entry, you will add a new URL parser package to the Go Standard Library. You might consider writing the code first. But you may know from experience with other languages that tests can help you write the correct code. Tests can also help you learn more about the Go testing framework that you may never have had the opportunity to work on before.

In this section, you will write your first unit test. At the end of the section, you'll have learned the basics of testing, written idiomatic tests, and code for parsing a URL. The url package won't be an exact replica of the Go Standard Library's url package. Still, it will show you writing idiomatic unit tests in a fun way.

2.2.1 Creating the url package

As you might recall from the chapter introduction, the Wizards team wanted to parse URLs using your url package. To do that, they would need to import the package. As you might already know, in Go, every directory corresponds to a single Go package. So you will start writing the url package in a new directory.

Let's make and switch to a new directory called *url* by entering the following commands at the command line:

```
mkdir url
cd url
```

You created a directory for the package, but you didn’t define it yet. You will do that in the next section, and your package will happily live in this directory.

Warning

About Go Modules

If you have downloaded the code from the book's github repository, you would already have the code for the url package in the ch02/url directory.

If you want to code along with the book and write code from scratch, you might wish first to create a new directory and initialize a new Go module using the following commands:

```
mkdir project_name
cd project_name
go mod init github.com/your_username/project_name
```

After you create the url package in this section, you can import like so:

```
import github.com/your_username/project_name/url
```

About naming of packages

Notice that you didn't call the package `url_parser` or `parsing` or `parse`. A package name should describe what it *provides*, not what it *does*. It may not be all about parsing a URL.

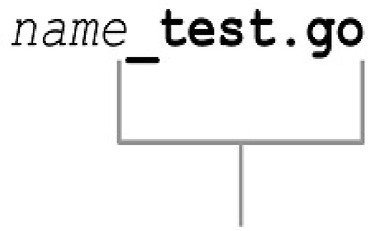
The url package provides ways for working with URLs. In the future, you can add more functionality to the url package, and you won't need to change the package's name.

However, there is not a single truth out there. You can do whatever is best for your own situation.

2.2.2 Creating a test file

Go has a special naming convention that makes a source code file a test file. As you can see in Figure 2.3, every test file should have a *_test.go* suffix so the test tool can see the file as a test file and automatically run it.

Figure 2.3 Naming of test files



A test file should end with the *_test.go* suffix

Since you're going to write a test for the `url` package, you can create a new *empty* test file called `url_test.go`. If you like, you can create the file at the command line by typing the following command:

```
$ touch url_test.go
```

Note

Windows does not have a touch command. Instead, you can create an empty file in your favorite editor.

Let's try running your first test file using the `go test` tool. While in the `url` directory, you can simply run the test using the test tool like so:

```
$ go test
expected 'package', found 'EOF'
```

You see this error because a test file is just another Go source-code file, and it also needs to belong to a package. It's better to keep things simple, especially at the beginning. You may not know yet what code to write, what to export, and what not to export from the url package. So instead of creating a new test package, you can put the code and tests in the same package (Listing 2.1).

Listing 2.1: The first test file (url_test.go)

```
package url
```

You can now access all the functionalities of the url package from tests, this is called an *internal test*. You will learn about *external* and *internal* tests later on.

You will see another error as follows when you run the test tool:

```
$ go test
testing: warning: no tests to run
```

This message comes from the test tool. It couldn't find any test functions in the package. You have a test file, but you don't have a test function, so there is nothing to run yet. No worries, you will create your first test function next.

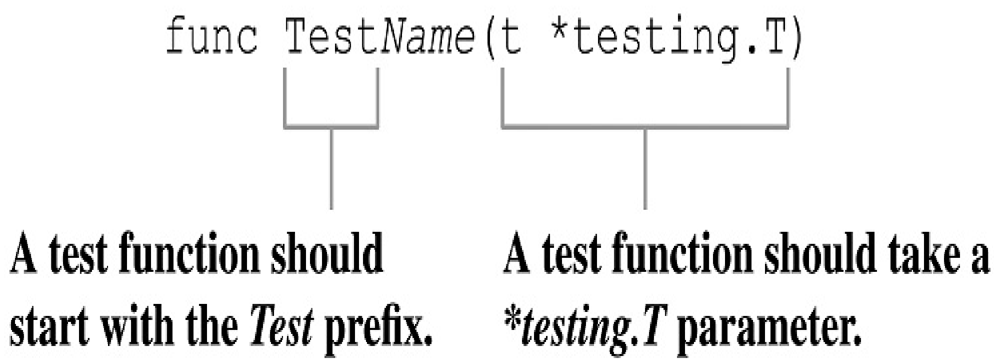
Exclusion of tests

If you're wondering about what happens to your tests when you compile your programs, read on. Only the test tool includes the tests in a compiled binary file. The compiler ignores files that end with the `_test.go` suffix when building your code with the `go build` command. So the tests won't end up inflating the binary file.

2.2.3 Writing a test function

In Go, you use simple functions for testing. As you can see in Figure 2.4, you write a function that begins with a Test prefix and takes a *testing.T parameter.

Figure 2.4 Naming of test functions



1. A test function should start with the `Test` prefix.
2. It should also take a `*testing.T` parameter.

Let's say you decided to write a function named `parse` that can parse a given URL. But you want to provide the parsing functionality to the Wizards team. To do that, you need to export the function from the `url` package by renaming it to `Parse`. That way, the Wizards team can call the function when they import the package. So you plan to have a `Parse` function in the end.

Tip

Even if you want to test an unexported (private) function, you would still write `TestParse` instead of `Testparse`. If you're worrying that there will be two tests with the same name, then you can make the other one `TestParseInternal`.

As I said earlier, before writing the `Parse` function, you will first write a test. You can see your first test function in listing 2.2.

Listing 2.2: The first test case (url_test.go)

```
package url

func TestParse() { #A
    // Nothing is here yet.
}
```

You named the test function as `TestParse` and put it into the url package. When you run the test tool, you get an error that says:

```
$ go test
wrong signature for TestParse, must be: func TestParse(t *testing
```

The test tool tells you that the signature of the test function is incorrect. A signature includes the name of a function, the parameters it takes and returns. You can make the test function correct by adding a `*testing.T` parameter to it. You can see the updated test code in listing 2.3.

Listing 2.3: A valid test case (url_test.go)

```
package url

import "testing"   #A

func TestParse(t *testing.T) {        #B
    // Nothing is here yet.
}
```

You named the test function as `TestParse`, have it take a `*testing.T` parameter, and put the test function into the url package. You also needed to import the testing package to use the `*testing.T` type. When you rerun the test, you get the following output:

```
PASS
```

Congratulations! You feel good that the test tool tells you that the test is successful. But you shouldn't feel like that because the test should have failed, right? Why did it succeed? It's because the testing package has no idea that your test is supposed to fail. To do that, you need to learn about the `*testing.T` type.

Note

There is only a single test function in listing 2.3, but you can write as many test functions as you want in the same test file or another. The test tool will automatically find and run them.

2.2.4 Testing by signals

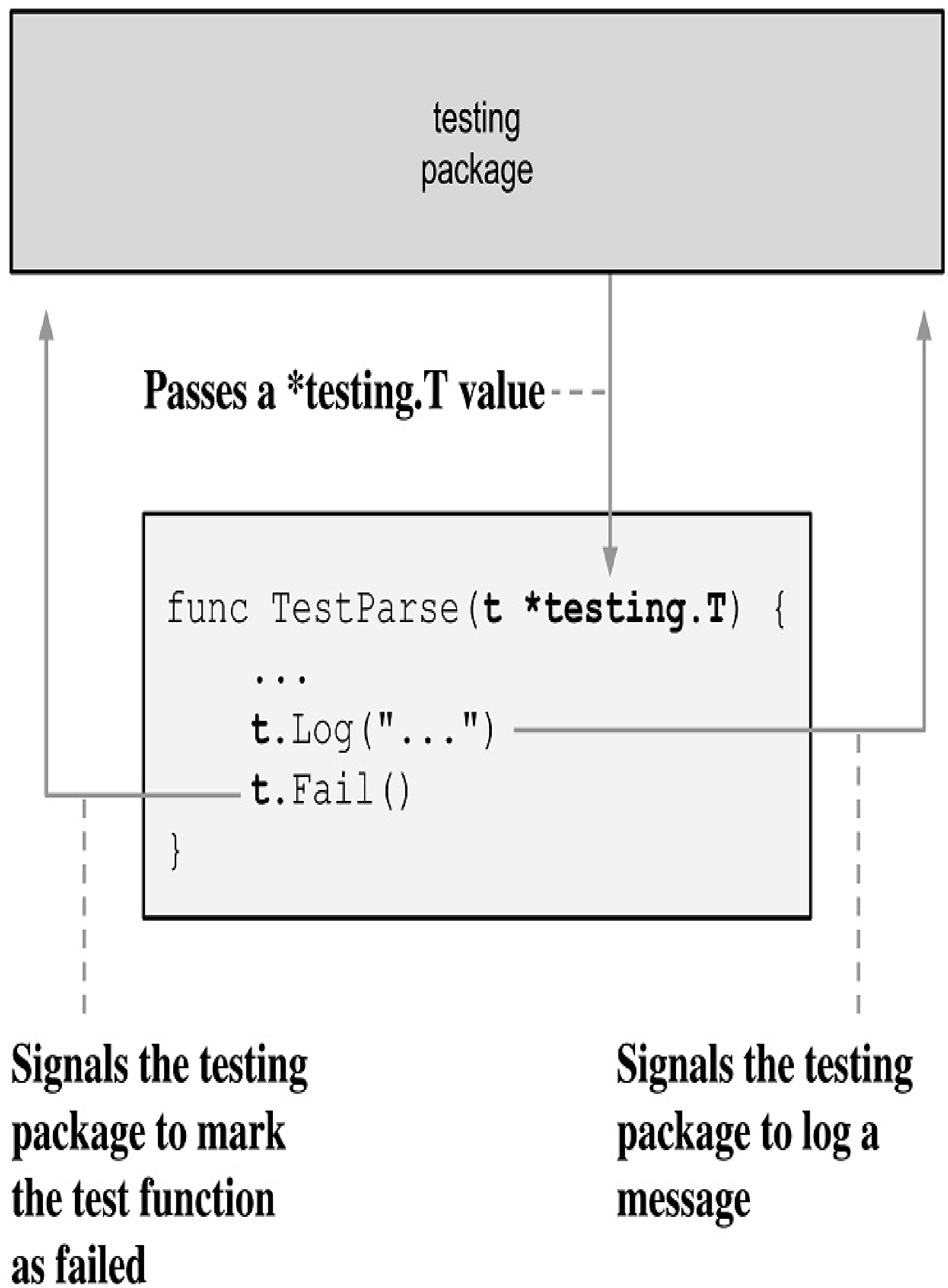
You finally have a valid test case. But it's not a helpful test case because it doesn't verify anything yet! In this section:

- You will learn how to communicate with the testing package using the `*testing.T` type.
- And how to write tests using the `*testing.T` type's signaling methods.

It would be a significant first step if you could write a failing test. So let's dig deeper into the `*testing.T` type and learn what methods it offers.

Let's say the testing package is running the `TestParse` test function. How is the test function going to *signal* to the testing package that it fails or succeeds? There needs to be a way to communicate with the testing package. For that, the testing package provides a couple of *signaling* mechanisms. One of them is the `*testing.T` type. As you can see in Figure 2.5, you can control the testing flow by communicating with the testing package.

Figure 2.5 The communication between the testing package and a test function


testing
package
Passes a *testing.T value
func TestParse(t *testing.T) {
...
t.Log("...")
t.Fail()
}
Signals the testing package to mark the test function as failed
Signals the testing package to log a message

1. The testing package passes a `*testing.T` value to a test function so that the test function can communicate with the testing package.
2. The testing package logs a message if a test function calls the `t.Log` method. The testing package logs the messages to the right below the test in the test output. So you can see that the messages are coming from that test.
3. The testing package marks a test function as failed if the test function calls the `t.Fail` method. Later on, it will report the function as failed in the test output.

Under the hood, the `*testing.T` type is a struct type in the Go Standard Library's *testing package*:

```
// src/testing/testing.go
package testing
type T struct {
    ...
}
```

Before running a test function, the testing package passes a `*testing.T` value. The `*T` type provides a couple of signaling methods, and you can see some of them in table 2.2. Throughout the chapter, you'll be learning about other signaling types as well. For now, you will only use the `Log`, `Logf`, and `Fail` methods.

Writing a failing test

You want to parse a given URL and get an error if the parsing fails. Otherwise, the Wizards team who will use the function will never know if the parsing fails. In the beginning, the parsing will always fail because you will be implementing the parsing logic later on. If you implemented the correct logic right away, you might not be sure that the test code would work correctly. So let's call the `Parse` function, get an error from it, and fail the test if there was an error.

In listing 2.4, your test function calls the url package's `Parse` function. Then if it gets an error from the `Parse` function, it sends a signal to the testing package using the `Fail` method. So the testing package marks the test as a failure.

Listing 2.4: Writing a failing test (url_test.go)

```
func TestParse(t *testing.T) {
    if err := Parse("broken url"); err != nil { #A
        t.Fail()   #B
    }
}
```

You might be sure you wrote a proper test function this time. But when you wanted to execute the test, you will get an error as follows:

```
$ go test
undefined: Parse
FAIL
```

The error message obviously tells you that the test failed. It's time to implement the `Parse` function next.

What happened here?

The test tool first tried to compile the code using the Go compiler, but it failed because you haven't yet declared the `Parse` function. That's why the test tool never reached the testing package due to the compilation error. Think about it. The test tool compiles tests with the testing package. So if the build failed, how would the testing package work?

Writing the code

You may have noticed that you haven't written the `Parse` function yet. That way, you saw what would happen when you ran the test without it. So let's get to it and first create a new empty file in the same directory. You're now ready to implement the `Parse` function in listing 2.5.

Listing 2.5: Writing the Parse function (url.go)

```
package url
import "errors"
```

```
// Parse parses rawurl into a URL structure.    #A
func Parse(rawurl string) error {     #B
    return errors.New("malformed url") #C
}
```

Tip

You might always want to document your function, mainly when you export it from a package. By the way, note that because the tests and code are in the same package, it is not necessary to export a function to test it.

When you run the test, it fails again:

```
$ go test
--- FAIL: TestParse
```

Congratulations! You're feeling great now that you've finally written a failed test. While it sounds like a joke, this test is a good milestone and allows me to explain more aspects of the testing package.

Putting code and tests in the same file

You could have put the parser code (`url.go`) into the same test file (`url_test.go`), but it's better to separate them. Otherwise, you may unintentionally mingle them, and that can easily lead to a maintainability nightmare. Sometimes, especially when prototyping, it's okay to put the code and tests in the same file.

Writing descriptive failure messages

When you read the previous output, you couldn't see why the test failed. Then you thought to yourself: "*I don't understand anything about this error message!*" I'm happy to say that there is a `Log` method that can print out a message. So if the test function fails (Listing 2.6), you can print the error from the `Parse` function.

Listing 2.6: Logging the error message (url_test.go)

```
func TestParse(t *testing.T) {
    if err := Parse("broken url"); err != nil {
        t.Log(err) #A
        t.Fail()   #B
    }
}
```

Now you have a test function that fails with an error message. When you run it, this time, it will fail with the error message that you get from the `Parse` function:

```
$ go test
--- FAIL: TestParse
    malformed url
```

While the current output looks better than the previous one, it is still not helpful enough (I know; I'm a picky person.) If you had a lot of test functions down the road, you wouldn't know why you were getting this error. So let's craft a better error message in Listing 2.7 using another method called `Logf`.

Listing 2.7: A better error message with the Logf method (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "broken url"

    if err := Parse(rawurl); err != nil {
        t.Logf("Parse(%q) err = %q, want nil", rawurl, err)
        t.Fail()
    }
}
```

As you can see in Listing 2.7, the `Logf` method acts like the `fmt.Printf` function. You first pass it a formatting specifier as a string, then a number of variadic arguments in any type. In Listing 2.7, the formatting specifier includes:

- `Parse(%q)`—This part of the message will log the function you are testing and the input it takes (The `%q` wraps a given string value by double-quotes.) So you can see which function has caused the test to fail and with what argument.
- `err = %q`—This part is about what error you get when the `Parse`

function returns so that you can look for this error in the `Parse` function's code and pinpoint the problem.
- `want nil`—This part shows what you wanted to see instead. You said "*want nil*" because you don't expect to get an error value. But if you get one, then it means something went terribly wrong.

When you run the test, you get a more descriptive error message:

```
$ go test
--- FAIL: TestParse
    Parse("broken url") err = "malformed url", want nil
```

Now the message tells you what URL you passed to the `Parse` function, the error you received from it, and what you wanted instead. With this error message, you can easily see why the test failed. Helpful, eh? No?

Passing the test

Up to this point, you have checked if your test function catches the error if the `Parse` function fails. If the parsing function successfully parses the URL, you should also check that the test was successful. So you can be sure that the test is working. You can see the code in listing 2.8.

Listing 2.8: Changing the Parse function to be successful (url.go)

```
func Parse(rawurl string) error {
    return nil
}
```

2.2.5 Wrap up

You created a new package called url and successfully wrote failing and passing tests for it! Good work! You learned that you could use the `Fail` method to fail a test. And how to write descriptive failure messages using the `Log` and `Logf` methods. Let's summarize what you've learned so far:

- Every directory corresponds to a single package.
- A package name should describe what it provides.
- A test file ends with a `_test.go` suffix and imports the testing package.

- A test function starts with a `Test` suffix and takes a `*testing.T` parameter. Naming your test functions after the function they test is a good practice. For example, `TestParse` tests the `Parse` function.
- The testing package passes a `*testing.T` value to each test function in a test file.
- Writing descriptive error messages helps you find why a test failed.
- `t.Log` logs a message to the test output.
- `t.Logf` logs a *formatted* message to the test output.
- `t.Fail` marks the test as failed, but the test keeps running. You'll soon find out why sometimes this is a disastrous thing but no worries because I have a solution for that too!

2.3 Writing a URL parser

Now that you've seen the basics of testing, it's time to start writing the parser code! The Wizards team asked you for a package to parse a URL, if you remember from the chapter introduction.

- They want to get the parts of a URL such as a scheme, host, and path. So when they receive a web request, they can check those fields and redirect the web request to a new location.
- To redirect a web request to a new URL, they want to change the parts of a URL. So your code will parse the URL and return a pointer to a parsed URL value. With a pointer, the developers can easily manipulate the parts of the URL.

Let's take a look at a simple URL and talk about its parts:

```
https://twitter.com/inancgumus
```

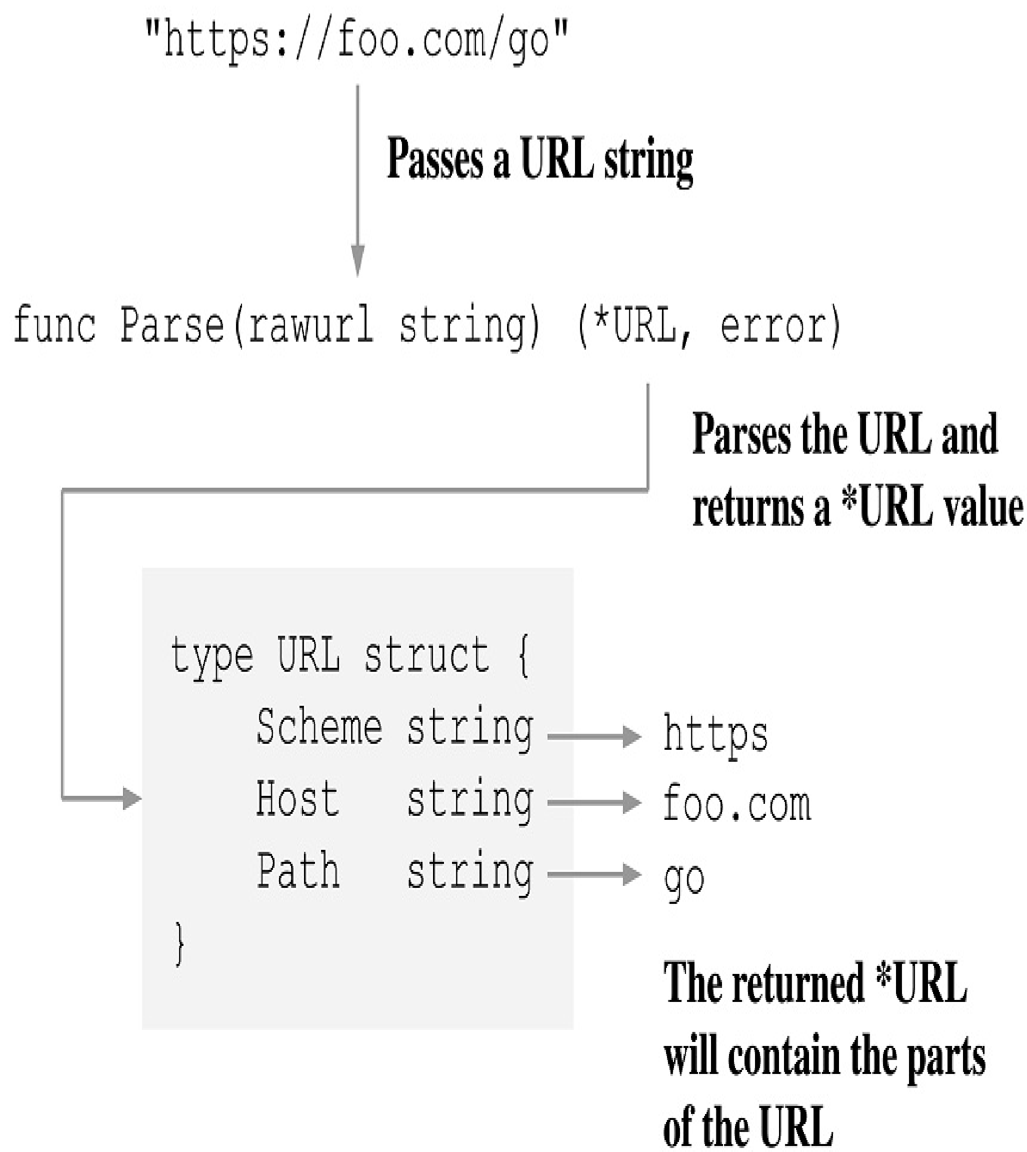
It looks as follows in abstract terms:

```
scheme://hostname/path
```

- *Scheme* is the protocol: *https*
- *Hostname* is the location of the resource on the Internet: *twitter.com*
- *Path* is the resource on the host: *inancgumus*

As you can see in Figure 2.6, you pass a URL string to the `Parse` function. Then, it parses a URL string, and it returns a pointer value of the URL struct. The URL struct type contains the URL parts, such as scheme, host, and path.

Figure 2.6 Parsing a URL



1. You first pass a raw URL as a string value to the `Parse` function.
2. Then the `Parse` function parses the raw URL and returns a pointer value of the `URL` struct.

3. If the `Parse` function can parse the raw URL, it returns a `nil` error. A `nil` error means everything went fine. Otherwise, it will return a specific error message that tells why it couldn't parse the raw URL.
4. The `*URL` value contains the parts of the raw URL. For example, let's say you pass `"https://foo.com/go"` to the `Parse` function. Then the function will return a `*URL` value with the `Scheme` field `"https"`, the `Host` field `"foo.com"`, and the `Path` field `"go"`.

About the parser

There are more things involved in a full-fledged parser, but you will keep things more manageable. So you will only parse the scheme, hostname, and path of a URL. There is no need to make things unnecessarily complex and make you bite your fingernails.

2.3.1 Parsing the scheme

In this section, you will begin with parsing the scheme of a URL. You will start by adding a new test case to the previous test function. After having a failing test, you'll write the necessary parser code to pass the test. In Table 2.3, you can see what you'll be parsing.

Table 2.3. URL scheme

Input: Raw URL	Output: Scheme
`https://foo.com`	https
`http://foo.com`	http

As you can see in Table 2.3, if the `Parse` function receives a string value that contains `https://foo.com` as a raw URL, it will fill the `Scheme` field of the URL type with `https`. Or if it receives `http://foo.com`, it will fill the

`Scheme` field with `http`.

Writing a test case

In your other favorite language, you might be writing a test case in three phases: *Arrange*, *Act*, and *Assert*. No worries, it's no different in Go. It's a good practice to write a test using the following phases:

- *Arrange*—You define some input values for the code that you want to test and your expectations of what you want the code to produce when you run it.
- *Act*—You run the code with the input values and save what you get from the code.
- *Assert*—You check whether the code works correctly by comparing the expected values with the actual values.

Note

The test in Listing 2.9 is intentionally broken (The variable `u` is missing).

Let's use this practice in action (Listing 2.9), and let's say you want to parse for `https://foo.com`.

Listing 2.9: Testing the URL scheme (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://foo.com"  #A

    if err := Parse(rawurl); err != nil {
        t.Logf("Parse(%q) err = %q, want nil", rawurl, err)
        t.Fail()
    }
    want := "https"          #B
    got := u.Scheme          #C
    if got != want {         #D
        t.Logf("Parse(%q).Scheme = %q; want %q", rawurl, got, wan
        t.Fail()   #F
    }
}
```

- *Arrange*—In this phase, you first set the stage for the expected outcomes. You want your test to return an `https` scheme. So you put this value in a variable called `want`. You could have used a constant instead, but you will change what you want throughout the same test function later on. That's why you used a variable instead.
- *Act*—In this phase, you actually get the scheme from the `Scheme` field and store the result in another variable called `got`.
- *Assert*—In this phase, you compare the variables: `want` and `got`. If they don't match then, it means that the `Parse` function didn't work correctly. In that case, you fail the test with a log message.

As you can see in Listing 2.9, you log a descriptive failure message if you get an error. But this time, you added `.Scheme` next to the `Parse(%q).` With that, you can quickly see that you're getting the `Scheme` field from what the `Parse` function returns.

The got-want naming convention

According to this idiom, a test *wants* something from the code it wants to verify. Then it checks to see whether it *gets* what it *wants*. Often, I prefer using this naming convention, but you don't have to use this convention at all. You can choose any convention you like. For example, you can use "exp" (short for expected) instead of "want."

Declaring the URL type

Since you didn't get a parsed URL from the `Parse` function, you will get an error when you run the test: `undefined: u`. So you will create a new type called `URL` and return it from the `Parse` function as a pointer:

1. For now, you only need the `Scheme` field in the `URL` type
2. You will declare the type in the `url.go` file, add the field, and return a new `*URL` value from the `Parse` function
3. Then you will change the `url_test.go` file and retrieve the parsed `*URL` from the `Parse` function

Let's first start with the `url.go` file and add the `URL` type in it, as you can see in listing 2.10.

Listing 2.10: Creating the URL type (url.go)

```
// A URL represents a parsed URL.     #A
type URL struct {  #B
    // https://foo.com       #C
    Scheme string // https   #D
}
```

Now, it's time to return the scheme from the `Parse` function. So, at first, you will not parse the scheme but will deliberately return a wrong scheme. You can see what you'll be writing in Listing 2.11.

Listing 2.11: Returning a URL with scheme (url.go)

```
func Parse(rawurl string) (*URL, error) {       #A
    return &URL{"fake"}, nil #B
}
```

Now the `Parse` function returns two values:

1. `*URL`: A pointer to a parsed `URL` value. In Listing 2.11, you return a `*URL` with a fake scheme.
2. `error`: In Listing 2.11, you return a `nil` error value because you don't want the `Parse` function to fail in the first part of the test function.

Since the `Parse` function returns one more value, you also need to change your test function. You will get the parsed URL and put in a variable called `u` in Listing 2.12.

Listing 2.12: Getting a parsed URL (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://foo.com"

    u, err := Parse(rawurl)  #A
    if err != nil {
    ...
    }
    ...
    got := u.Scheme
```

```
    ...
}
```

The test function in Listing 2.12 looks good. It parses a rawurl and gets the parsed URL. Then it gets the `Scheme` field from the parsed URL. Now, it's time to execute your new test. The output will look similar to the following when you run the test:

```
$ go test
--- FAIL: TestParse
    Parse("https://foo.com").Scheme = "fake"; want "https"
```

The message tells you a few things about why the test failed:

- You passed a raw URL to the `Parse` function: `"https://foo.com"`
- You wanted the `Scheme` field to be `"https"`
- But you received a `fake` scheme instead

Congratulations! You verified that the test works, and the failure message looks descriptive. What more could a homo sapiens programmer want?

Writing the parser code

You might be sure that the new test works. So now you want to parse the scheme from the URL instead of returning a fake scheme. You will write simple parser code that will split the raw URL by this value `"://"`. You will first change the `url.go` in Listing 2.13 and then execute the test to be sure that it works.

Listing 2.13: Parsing the scheme (url.go)

```
...
import (
    "strings"
    ...
)
...
func Parse(rawurl string) (*URL, error) {
    i := strings.Index(rawurl, "://") #A
    scheme := rawurl[:i]     #B
    return &URL{scheme}, nil #C
```

```
}
```

Note

Please don't forget to import the `strings` package.

- The `strings.Index` function finds the position of a string inside another string. In Listing 2.13, you look for a scheme ending in the `rawurl` and save it to a variable called `i`.
- Once you get the scheme's ending position, you slice the `rawurl` up to the scheme's ending position.
- And return the scheme packed in a new `*URL` value.

The test will pass when you run it. Wonderful! Now you completed one of your tasks, and you are getting closer to achieving the URL parser. Exciting, isn't it?

Fixing the parser code

Let's say you were poking up the test you just wrote and realized a fatal problem within the `Parse` function. With childlike excitement, you wondered what would happen if you passed a malformed url to the `Parse` function without a scheme. How about playing with that now? So let's change the `rawurl` as follows:

```
func TestParse(t *testing.T) {
    const rawurl = "foo.com"
    ...
}
```

When you tested it, the output looked similar to the following:

```
$ go test
--- FAIL: TestParse (0.00s)
panic: runtime error: slice bounds out of range [:-1] [recovered]
goroutine 6 [running]:
testing.tRunner.func1.2(0x1134620, 0xc00001c1b0)
/usr/local/go/src/testing/testing.go:1144 +0x332
...
```

The test panicked. But what's the problem? Let's check out Listing 2.13, where you used a function called `strings.Index`.

- You passed it a raw URL without a scheme.
- So it couldn't find `"://"` in the raw URL and returned minus one.
- Then you tried to slice the raw URL with a negative value, and the Go runtime didn't allow you to do that and panicked.

You can fix the problem quite easily. You can return an error if the `strings.Index` function cannot find the scheme pattern in the raw URL. You can see the fix in listing 2.14.

Listing 2.14: Fixing the parser (url.go)

```
func Parse(rawurl string) (*URL, error) {
    i := strings.Index(rawurl, "://")
    if i < 0 {     #A
        return nil, errors.New("missing scheme")          #A
    } #A
    scheme := rawurl[:i]
    ...
}
```

- If the `rawurl` doesn't contain the scheme separator ("`://`"), the `strings.Index` will return `-1`.
- So the if statement will catch it and return an error.
- With this change, you won't be slicing the `rawurl` with a negative value.

Panicking test functions

A panic always belongs to a single Goroutine, and that's why the Go runtime also prints the panicking Go routine with its stack trace. If you're coding along, you can find the origin of the error if you analyze the stack trace on your machine.

The Fail method

But when you ran the test, you came across another problem:

```
$ go test
--- FAIL: TestParse
    Parse("foo.com") err = "missing scheme", want nil
    panic: runtime error: invalid memory address...
```

You saw that the test caught the parsing error and showed the *missing scheme* error:

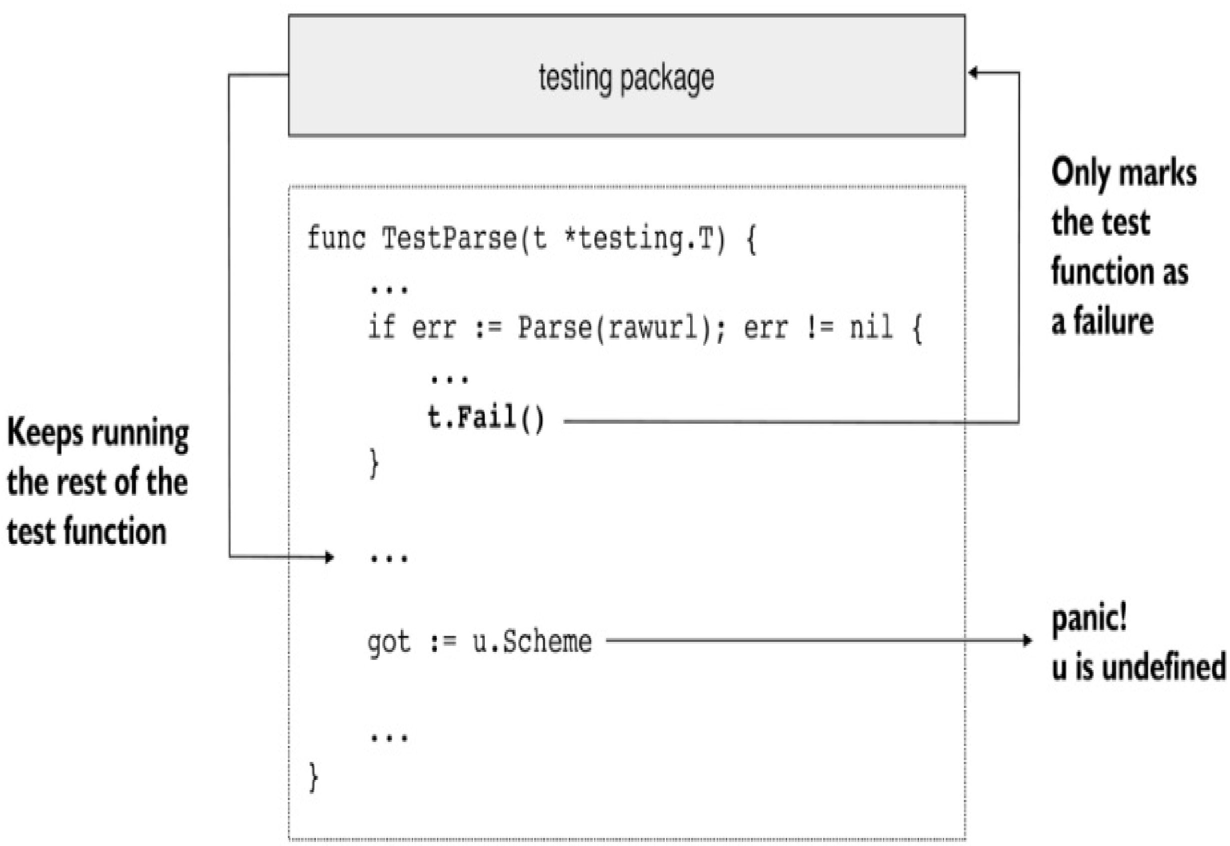
```
Parse("foo.com") err = "missing scheme", want nil
```

But the next error message is more interesting:

```
panic: runtime error: invalid memory address...
```

The test panicked again! But why? Remember that you return a `nil *URL` value if a raw URL doesn't contain a scheme (Listing 2.14). Let's take a look at the test function in Figure 2.7.

Figure 2.7 The Fail method does not stop a test function



1. You can see that the test function tries to get the `Scheme` field from a `nil *URL` value, resulting in a panic within the test function.
2. This problem occurs because the `Fail` method does not stop a test function.
3. So the test function keeps working even with a `nil *URL`!

If you can stop the test function from running when it gets an error from the parsing function, you can fix this problem. The solution is as easy as it sounds.

The FailNow method

Instead of calling the `Fail` method, you need to use another method called `FailNow` from the `*T` type. Let's learn about the implementation of the `FailNow` method to understand the inner workings of it better. As you can see in Listing 2.15, the `FailNow` method fails a test function and immediately stops its execution.

Listing 2.15: The FailNow method of the *T type

```
// src/testing/testing.go

package testing

// FailNow marks the function as having failed and stops its exec
// by calling runtime.Goexit (which then runs all deferred calls
// current goroutine).
func (c *T) FailNow() {      #A
    c.Fail()       #B
    ...
    runtime.Goexit()        #C
}
```

- In Listing 2.15, the `FailNow` method already calls the `Fail` method.
- So it marks the test as a failure.
- Then it calls the `runtime.Goexit` function so that the Go runtime stops running the test function.

Tip

You can find the source code of the Go Standard Library at this link:
https://github.com/golang/go/tree/master

The testing package runs each of your test functions in a separate Goroutine. Even if a test function calls `FailNow` and terminates, the other test functions will keep running (The test functions can be in the same test file or another). The testing package will also catch when a test function ends and report it in the test summary.

So let's use the `FailNow` method instead of the `Fail` method in the test function. You can see the code in Listing 2.16.

Listing 2.16: Using the FailNow method (url_test.go)

```
func TestParse(t *testing.T) {
    ...
    u, err := Parse(rawurl)
    if err != nil {
        ...
        t.FailNow()          #A
    }
    ... #B
}
```

When you run the test, the output should look as follows:

```
$ go test
--- FAIL: TestParse
    Parse("foo.com") err = "missing scheme", want nil
```

As you can see, now the test failed, and you didn't get a panic error. The test function stopped running after you called the `FailNow` method. With that, you avoided the code from trying to get the `Scheme` field when the parsed URL is nil. You often want to stop a test function when a fatal error occurs, or there is no need to continue running the test function.

The Fatal and Fatalf methods

Let's say you're also working on another project and writing a lot of test functions. But you're tired of calling logging and error functions every time you want to test something. I have good news for you! As you can see in Table 2.4, there are better ways. The testing package provides `Fatal` and `Fatalf` methods for logging and ending a test at the same time.

Table 2.4. The Fatal methods

Method	Purpose
`t.Fatal`	It is equivalent to calling Log and FailNow methods
`t.Fatalf`	It is equivalent to calling Logf and FailNow methods

When a test function calls the `Fatal` method, the testing package will log a message and immediately stop the test function by calling the `FailNow` method. Just like the `Log` method, the `Fatal` method takes variadic `any` parameters (`...any`). So you can call it with any type and number of values. It will log these values when you call the method.

The only difference between the `Log` and `Fatal` methods is that the `Fatal` method also stops running the test function.

The `Fatalf` method is similar to the `Fatal` method, but it also allows you to pass a formatting specifier with the `format` parameter. So you can use it like the `Logf` method.

Now that you understand what the `Fatal` and `Fatalf` methods do, you can replace the `Logf` and `FailNow` methods in the test function with a single `Fatalf` method in Listing 2.17.

Listing 2.17: Using the Fatalf method (url_test.go)

```
func TestParse(t *testing.T) {
    ...
    if err := Parse(rawurl); err != nil {
```

```
        t.Fatalf("Parse(%q) err = %q, want nil", rawurl, err)

        // t.Fatalf is the same as calling the following methods:
        // t.Logf("Parse(%q) err = %q, want nil", rawurl, err)
        // t.FailNow()       #B
    }
    ... #C
}
```

When you run the test, you will have the same output as before:

```
$ go test
--- FAIL: TestParse
    Parse("foo.com") err = "missing scheme", want nil
```

You didn't use the `FailNow` method in Listing 2.17 because you wanted to print a descriptive error message as a friendly gopher. You use the `FailNow` method if you want only to fail the test function without logging a message. In the end, thanks to the `Fatalf` method, you replaced two method calls with a single method call. Doing so made the code concise and more readable.

The empty interface type and ellipses

Before learning about the empty interface type, let's take a look at the implementations of the `Fatal` and `Fatalf` methods as they use the empty interface type:

```
// Fatal is equivalent to Log followed by FailNow.
func (c *T) Fatal(args ...any) {
    c.log(fmt.Sprintln(args...))
    c.FailNow()
}
```

It takes variadic any parameters, logs a failure message, and ends the caller test function. The `Fatalf` method is similar to the `Fatal` method but it also adds a *formatter* parameter for the failure message:

```
// Fatalf is equivalent to Logf followed by FailNow.
func (c *T) Fatalf(format string, args ...any) {
    c.log(fmt.Sprintf(format, args...))
    c.FailNow()
}
```

The `any` type can represent any type—hence the name.

It's actually an interface type. Go 1.18 renamed the empty interface type (`interface{}`) to the `any` type. They are the same types!

`interface{}` means an interface type without any methods, so it's literally empty but useful! This means that any type in Go can satisfy this empty interface type. You can think of it as the `Object` type from Java (or Javascript) or `object` type from Python.

For example:

```
var anything interface{}
// Or (same as above):
// var anything any
anything = 3
anything = "three"
anything = []string{"let", "there", "be", "light"}
```

The ellipses (`...`), on the other hand, make a function accept a variable number of arguments. So a function that has a `...any` parameter can accept any type of an arbitrary number of values.

The Error and Errorf methods

You've refactored the testing function using a more useful `Fatalf` method. Similarly, it is cumbersome to call logging and error functions every time you want a test to fail. As you can see in Table 2.5, fortunately, the testing package provides two more methods called `Error` and `Errorf,` both for logging and failing a test at the same time.

Table 2.5. The Error methods

Method	Purpose
`t.Error`	It is equivalent to calling Log and Fail methods.

`t.Errorf`	It is equivalent to calling Logf and Fail methods.

In Listing 2.18, you can see the implementation of the `Error` method. It takes variadic `any` parameters (`...any`), so you can pass any type and number of values to the method. Like the `Fatal` method, the `Error` method first logs an error message then fails a test function. The difference between them is that the `Error` method only marks a test function as a failure and keeps running the test function.

Listing 2.18: The implementation of the Error method

```
// src/testing/testing.go
package testing
// Error is equivalent to Log followed by Fail.
func (c *T) Error(args ...any) {      #A
    c.log(fmt.Sprintln(args...))      #B
    c.Fail()      #C
}
```

In Listing 2.19, you can see the implementation of the `Errorf` method. It takes a formatting specifier and variadic `any` parameters (`...any`).

Listing 2.19: The implementation of the Errorf method

```
// src/testing/testing.go
package testing
// Errorf is equivalent to Logf followed by Fail.
func (c *T) Errorf(format string, args ...any) {         #A
    c.log(fmt.Sprintf(format, args...))         #B
    c.Fail()      #C
}
```

Similar to the `Error` method, the `Errorf` method keeps running a test function even though the test function fails. It first logs an error message then marks a test function as a failure. The difference from the `Error` method is that the `Errorf` method also takes a formatting specifier. So it allows you to print descriptive failure messages.

Refactoring the test

Now it's time to refactor the test you wrote. Let's replace the `Logf` and `Fail` methods in the test function with a single `Errorf` method (Listing 2.20). As you've learned, behind the scenes, the `Errorf` method will call the `Logf` and `Fail` methods for you. Since you want formatted output, you didn't use the `Error` method and used `Errorf` instead (Listing 2.20). You got rid of two method calls with a single `Errorf` method call and achieved the same outcome.

Listing 2.20: Calling the Errorf method (url_test.go)

```
func TestParse(t *testing.T) {
    ...
    if got != want {
        t.Errorf("Parse(%q).Scheme = %q; want %q", rawurl, got, w

        // t.Errorf is the same as calling the following methods:
        // t.Logf("Parse(%q).Scheme = %q; want %q", rawurl, got,
        // t.Fail()          #B
    }
    // ...         #C
}
```

Let's take a look at the final test function in Listing 2.21. Did you notice that the `got` variable is next to the if-statement? Doing so is a good practice because the test function uses the variable only in the if-statement, making the code concise! You can see the final test function in Listing 2.21.

Listing 2.21: Calling the Errorf method (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://foo.com"

    u, err := Parse(rawurl)
    if err != nil {
        t.Fatalf("Parse(%q) err = %q, want nil", rawurl, err)
    }
    want := "https"
    if got := u.Scheme; got != want { #A
        t.Errorf("Parse(%q).Scheme = %q; want %q", rawurl, got, w
    }   #A
```

```
}
```

When you run the test, it will pass, and it won't panic. If you provide a URL without a scheme, the test function will stop running using the `Fatalf` method. So the test function won't try to get the scheme from a `nil *URL` value. The test function will only keep running if the `Parse` function can parse the URL. If it succeeds but cannot parse the scheme, the test output will mark the test as a failure and report an error using the `Errorf` function.

Wrap Up

Well done! You now have a URL parser with idiomatic tests. Of course, you are not done yet. In the next section, you will parse the hostname of a raw URL. But before moving on to the next section, let's discuss what you've learned so far.

You wrote test cases with the phases of arranging, acting and asserting. You set your expectations during the arrange phase, run the code under test during the act phase, and finally, if you get an undesired result in the assert phase, you mark the test as a failure:

```
want := "https"   // arrange
got := u.Scheme   // act
if got != want {} // assert
```

You also learned about the *got-want* naming convention where you put the test expectation in a variable called `want`, and you run the code and put the outcome in a variable called `got`. As I said before, it's just a naming convention, and you can invent your own way as long as you and other developers are comfortable using it.

You also learned when to use some of the testing methods. You use the fatal methods for ending a test function so that the remaining test function will stop running. But, if you want the test function to keep running, you use the error methods. You can find a summary of all the test methods that you've learned in table 2.6.

Table 2.6. The *testing.T methods used so far

Method	Purpose
`t.Log(args ...any)`	Logs a message to the test output
`t.Logf(format string, args ...any)`	Logs a formatted log message to the test output
`t.Fail()`	Marks the test as a failed test and keeps running the test function
`t.FailNow()`	Marks the test as a failed test and stops running the test function
`t.Error(args ...any)`	It is equivalent to calling Log and Fail methods
`t.Errorf(format string, args ...any)`	It is equivalent to calling Logf and Fail methods
`t.Fatal(args ...any)`	It is equivalent to calling Log and FailNow methods
`t.Fatalf(format string, args ...any)`	It is equivalent to calling Logf and FailNow methods

You also lifted the curtain a little bit to see how the testing package runs test functions. You learned that the testing package runs each test function in a separate Goroutine. So the testing package can still catch and report an error even if one of the test functions panics.

2.3.2 Parsing the hostname

In the previous sections, you parsed the scheme from a URL. The Wizards team wanted to analyze the host of a URL too. So that they can redirect web requests when the requests come from a specific set of domains. In this section, you will parse the hostname of a URL. So when you pass `"https://foo.com"` to the `Parse` function, it will return a `*URL` value with the `Scheme` field `"https"`, and `Host` field `"foo.com"`. You can see what you'll be parsing in table 2.7.

Table 2.7. URL hostname

Raw URL	Scheme	Host
`https://foo.com`	https	foo.com

Writing the test case

Let's get started and add a new test case to the existing test function. As you can see in Listing 2.22, the new test case is similar to the previous one (Listing 2.21). First, you get the `Host` field (*which does not exist yet*) and then compare it to the hostname. You mark the test as a failure if the expected hostname doesn't match the parsed one.

Listing 2.22: Testing for the hostname (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://foo.com"
    ...
```

```
    if got, want := u.Host, "foo.com"; got != want {
        t.Errorf("Parse(%q).Host = %q; want %q", rawurl, got, wan
    }
}
```

Tip

`` `if variable := value; condition` `` is called a short-if declaration. You can use it to declare the variables in the scope of the if statement. Declaring variables in a shorter scope is a good practice that helps avoid polluting the scope namespace with unnecessary variables.

When you run the test, you will get an error that tells you that the `Host` field does not exist yet:

```
$ go test
u.Host undefined (type *URL has no field or method Host)
```

This error validates that the compiler can compile the test and the only error is the missing `Host` field.

Writing the parser code

You will first add the `Host` field to the URL type and then write the parsing logic in the `Parse` function. You can see the updated URL type in Listing 2.23.

Listing 2.23: Adding the Host field to the URL (url.go)

```
type URL struct {
    // https://foo.com
    Scheme string // https
    Host   string // foo.com
}
```

Now you have the necessary fields, let's begin writing the parsing logic for the `Host` field. You can see the updated code in Listing 2.24.

Listing 2.24: The Parse function (url.go)

```
func Parse(rawurl string) (*URL, error) {
    i := strings.Index(rawurl, "://")
    ...
    // scheme := rawurl[:i]  #A
    scheme, host := rawurl[:i], rawurl[i+3:]    #B
    return &URL{scheme, host}, nil    #C
}
```

In the current version of the `Parse` function, you find the starting index of the scheme in the raw URL. So the rest of it contains the hostname that you're looking for. You can easily get the hostname by moving beyond the starting position of the scheme plus three. It's because you expect a scheme to end with the `"://"` pattern. You can see the test function passes if you run it.

Great! The Wizards team can get the hostname of a URL to redirect a request to a new domain or stop accepting traffic from that specific domain.

2.3.3 Parsing the path

You added the url package to the Go Standard Library and let the Wizards team know about this new change. They started using the package, but they also wanted to analyze the path of a URL. So that they can accept or deny access to a path. Or they can change the path of a URL and redirect a web request to a new location. In this section, you will parse the path of a URL. Let's take a look at what you'll be parsing in Table 2.8.

Table 2.8. URL path

Raw URL	Scheme	Host	Path
`https://foo.com`	https	foo.com	""
`https://foo.com/go`	https	foo.com	go

Tip

`""` is called an empty string in Go. It's not the same as `null` or `undefined`, unlike some other languages. It's a valid string value and its length is zero (`len("")` is equal to 0). A string variable can have an empty string value and you can still take its memory address. For example: `e := ""`. Then: `p := &e`. The variable `p` will point to the memory address of `e`.

As you can see in table 2.8, when a raw URL does not have a path, you will return an *empty string*. Otherwise, you will parse the path from the raw URL and put it into a new field called `Path` in the URL type. As usual, you will start with a new test case. Then, you will add a `Path` field to the URL type and modify the parser accordingly.

Revisiting the parser code

Previously in Listing 2.24, you only parsed the scheme and hostname from a raw URL. Now, you're expecting a path as well that begins after a hostname. For example, in `"https://foo.com/go"`, the path is `"go"`. But the current parser cannot separately parse the hostname and path. Let's try how the parser reacts by changing the `rawurl` in the `url_test.go` as follows:

```
const rawurl = "https://foo.com/go"
```

You will see the following error when you run the test:

```
$ go test
Parse("https://foo.com/go").Host = "foo.com/go"; want "foo.com"
```

As you can see from the failure message, the current parsing logic is parsing the hostname together with the path. It puts the path into the `Host` field. So you need to find a way to parse the path separately from the hostname. To do that, you will store the rest of the raw URL in a new variable called `rest`. And then, you will parse the hostname and path from that variable.

For now, let's first parse the hostname if there is a path in a raw URL. You can see the updated code in Listing 2.25. With this change, the test will pass if you run it.

Listing 2.25: Parsing the path (url.go)

```
func Parse(rawurl string) (*URL, error) {
    ...
    scheme, rest := rawurl[:i], rawurl[i+3:]    #A
    host := rest   #B
    if i := strings.Index(rest, "/"); i >= 0 {  #C
        host = rest[:i]      #D
    }   #C
    return &URL{scheme, host}, nil
}
```

Writing the test case

Now your new parser can parse the hostname whether there is a path in a raw URL or not. So finally, it's time to parse the path. Let's first add a new test case to your test function that you can see in Listing 2.26.

Listing 2.26: Testing for the path (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://foo.com/go"         #A
    ...
    if got, want := u.Path, "go"; got != want { #B
        t.Errorf("Parse(%q).Path = %q; want %q", rawurl, got, wan
    }
}
```

Writing the parser code

The test in Listing 2.26 will fail if you run it because you don't have a `Path` field in the URL type yet. Listing 2.27 adds the `Path` field to the URL type.

Listing 2.27: Adding the Path field (url.go)

```
type URL struct {
    // https://foo.com/go
    ...
    Path   string // go
}
```

Now you have the `Path` field in the URL type; it's time to parse the raw URL path. You can see the updated code in Listing 2.27.

Listing 2.28: Parsing the path (url.go)

```
func Parse(rawurl string) (*URL, error) {
    ...
    host, path := rest, ""   #A
    if i := strings.Index(rest, "/"); i >= 0 {
        host, path = rest[:i], rest[i+1:]       #B
    }
    return &URL{scheme, host, path}, nil        #C
}
```

With this change, the new parser can work with or without a path. And, you only parse the path if there is one in the raw URL. Well done! The test will pass if you run it.

About multiple assignments

In Go, you can assign multiple variables at once. For example, if you want to assign the `rest` variable to the `host` variable and an empty string to the `path` variable, you can do it as follows:

```
host, path := rest, ""
```

The code above is equivalent to the following:

```
host := rest
path := ""
```

For more information about all the rules about multiple assignments, see my posts at the links: https://stackoverflow.com/questions/17891226/difference-between-and-operators-in-go/45654233#45654233 and https://blog.learngoprogramming.com/learn-go-lang-variables-visual-tutorial-and-ebook-9a061d29babe

2.3.4 Wrap up

Let's summarize what you've learned in this section so far:

- It's a good practice to write a test in Arrange, Act, and Assert phases. The Arrange phase defines some input values for the code that you want to test and your expectations of what you want the code to produce when you run it. The Act phase runs the code with the input values and saves what you get from the code. The Assert phase checks whether the code works correctly by comparing the expected values with the actual values.

- The *got-want* is the most commonly used naming convention for testing in Go.
- A panic always belongs to a single Goroutine, and the testing package runs each test function in a different Goroutine.
- A test function immediately stops and fails when it calls the `FailNow` method. Even if a test function calls `FailNow` and terminates, the other test functions will keep running.
- The `Fatal` method is equivalent to calling the `Log` and `FailNow` methods.
- The `Fatalf` method is equivalent to calling the `Logf` and `FailNow` methods.
- The `Error` method is equivalent to calling the `Log` and `Fail` methods.
- The `Errorf` method is equivalent to calling the `Logf` and `Fail` methods.

2.4 Summary

Wow! That was quite an adventure! You learned a lot of things about writing idiomatic code and testing in Go. Well done! I think the best way to learn is by doing. I hope you coded along while reading the chapter. As your tests grow, they will become complex. It will be hard to manage them. In the next chapter, you will also learn how to tame the complexity monster.

- The test framework is versatile and straightforward. The test tool finds tests and prepares the necessary environment. The testing package runs the tests and reports their results in summary.
- A test filename ends with the `_test.go` suffix. A test function starts with the `Test` prefix and takes a `*testing.T` parameter. The `*testing.T` type allows you to communicate with the testing package.
- The `Error` method is equal to calling the `Log` and `Fail` methods. But it doesn't stop a test function. The `Fatal` method is equal to calling the `Log` and `FailNow` methods. It does stop a test function when a fatal error occurs.
- Writing descriptive failure messages allows you to pinpoint the cause of

a problem without looking at the source code.

3 Fighting with Complexity

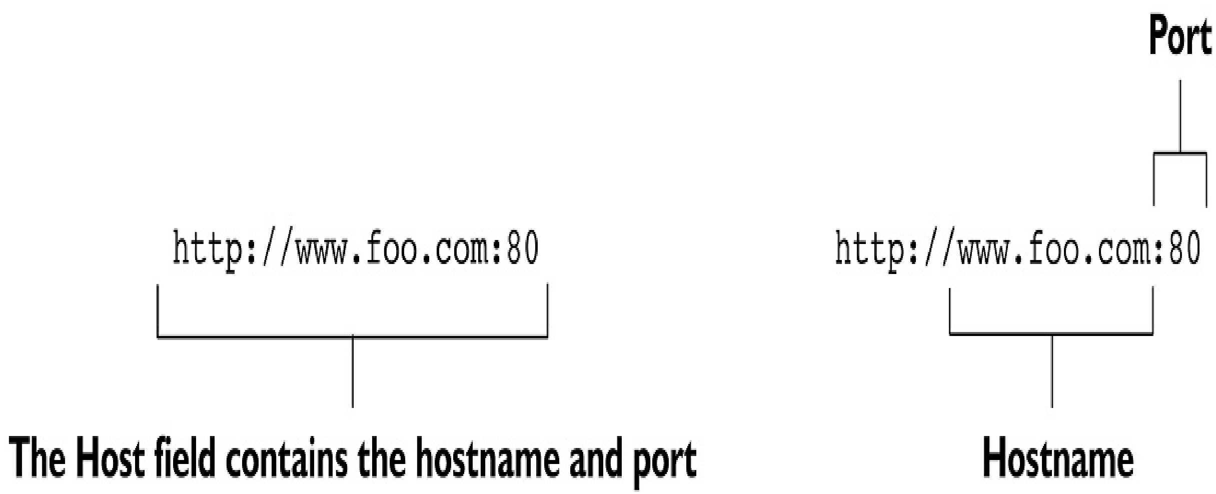
This chapter covers

- Reduce repetitive tests using table-driven testing
- Run tests in isolation using subtests
- Learn the tricks of writing maintainable tests
- Learn to shuffle the execution order of tests
- Parse port numbers from a host

Do you remember the Wizards team? They need a new feature! You might ask: "What's wrong with these people? They are so demanding! But isn't that the entire purpose of our profession?" saving people, hunting bugs, the family business. So let's see what they're asking.

They're receiving web requests with port numbers and need to get the *hostname* and *port number* from a URL to deny access to some hostnames and ports. They tried to get these values using the `url` package you wrote earlier, but it didn't help as the `Host` field of the `URL` type stores hostnames along with port numbers (Figure 3.1).

Figure 3.1 The Wizards team needs to get the hostname and port number of a URL, but the url package doesn't offer a way.



They submitted a proposal to the Go repository on Github and asked for help. Imagine you responded to their proposal and started working on the problem. But you realized that you're writing a lot of repetitive test functions.

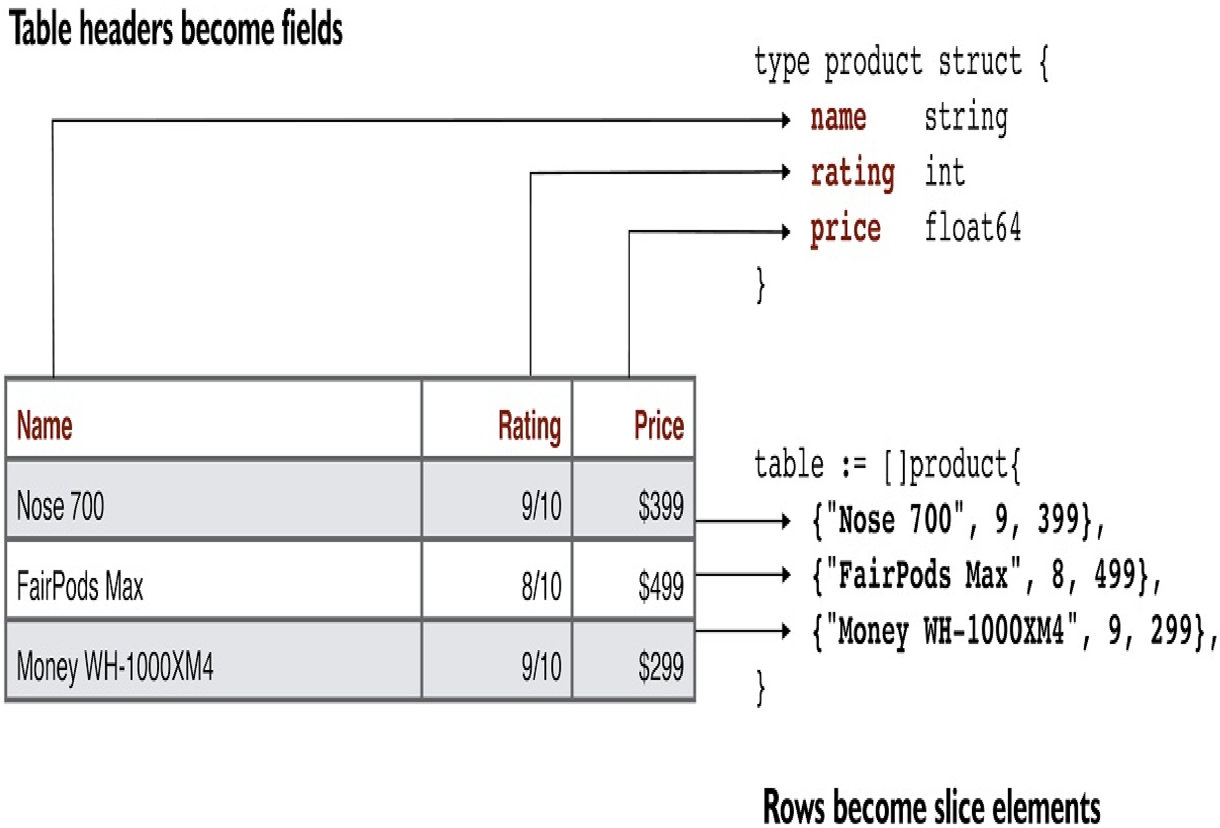
Is there a better way? Well, there is something called table-driven testing (*also called Data-Driven Testing and Parameterized Testing*)! It is for verifying variations of the same code with varying inputs. There are also subtests for running tests in isolation. In this chapter, I'll help you beat complexity using table-driven tests and subtests.

3.1 Table-Driven Testing

Data tables are everywhere in our lives and make our lives more manageable. Imagine you want to buy those shiny new noise-canceling headphones to cut out the background noise and write excellent code (*I know, sometimes it's just for procrastination purposes.*)

So you go and start researching headphones on the Internet and create a table in a spreadsheet program like the one in Figure 3.2. So you can better decide which headset you want to buy. That's why people say it's a *decision-table*. It makes it easy to do a *"what-if"* type of analysis.

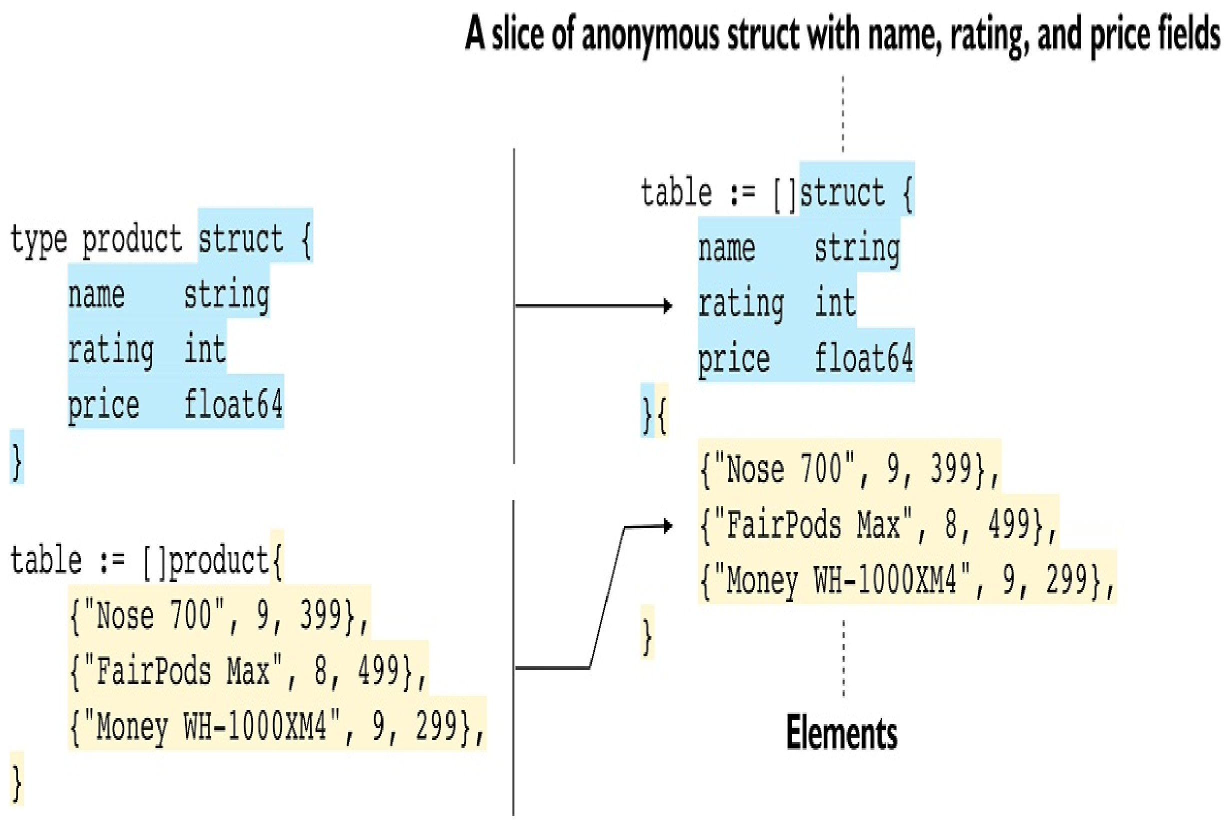
Figure 3.2 Expressing a table in code



Since a headphone is a *product* you want to buy, why not declare a new struct called `product` as in Figure 3.2? The `product` struct is a *description* but it contains nothing but *metadata*. So you have one more step to take. You can create a *slice* from the `product` structs and put the headphone data in it (Figure 3.2).

You can think of the `table` slice as the table in Figure 3.2, expressed by code. It has three elements with information about the headphones. You can also express the table more concisely from scratch, as in Figure 3.3.

Figure 3.3 Anonymous structs do not have a name



The struct in Figure 3.3 is called an *anonymous struct* and does not have a name. It is for one-time use only, so it disappears after using it. I mean, after declaring the table variable, you can no longer use it. You will soon see that we gophers often prefer to write table-driven tests this way.

Now that you understand what a table looks like and why it can help describe data, it's time to get your feet wet. In this section, you will write your first table-driven test. First, let's try writing the tests you learned in the previous chapters before writing them in the table-driven style. That way, you can see why and where you might need table-driven testing.

3.1.1 Parsing port numbers

The Wizards team wants to get the hostname and port number from a URL. To keep things simple for now, you'll parse the hostname later. So how about starting by adding a `Port` method to the `URL` type and getting the port number with it?

Note

The `Port` method could be a function rather than a method, but it's better to add it to the `URL` type. So one can quickly get the port from a parsed URL.

To do this, you can start writing a test function called `TestURLPort` in Listing 3.1 to verify whether you can get a port number from the host. You will get a port number from the host `"foo.com:80"`. It doesn't matter as long as the host has a port number. This way, you can see whether you can parse the port number. So the test will make a new `URL` value for the host.

Listing 3.1: TestURLPort (url_test.go)

```
func TestURLPort(t *testing.T) {
    const in = "foo.com:80"  #A

    u := &URL{Host: in}
    if got, want := u.Port(), "80"; got != want {         #B
        t.Errorf("for host %q; got %q; want %q", in, got, want)
    }
}
```

You may have noticed that the test won't pass because you haven't implemented the `Port` method yet. So, for now, let's create the new method but have it return an *empty string* so you can see the error message when the test fails:

```
func (u *URL) Port() string {
    return ""
}
```

About adding a Port field to the URL type

Here are some questions for you:

· Can't you just add a new field called `Port` to the `URL` type?

· Why are you adding a new method to the `URL` type instead?

Before answering these questions, let's remember the `URL` type:

```
type URL struct {
    Scheme string
    Host   string
    Path   string
    // Should you add a Port field here?
}
```

If you added a `Port` field to the URL type, what does the `Host` field store? A host without or with a port? If it were the former, you would change the behavior of the url package and break any code depending on the package.

Many people may be using the `url` package and `Host` field in their programs. But, for a URL with a port, they assume that the `Host` field contains a port number (before adding the `Port` field.) So instead of changing the behavior of the `Host` field, it's better to add a method that returns a port from the `Host` field.

You can invent more creative solutions to this problem, however, our current approach is needed because of Go 1.0 backward guarantee. And, we're trying to add a package to the Go Standard Library. You might be (or will be) having similar problems as well. See the article at the link for more information: https://go.dev/doc/go1compat.

Let's discuss the error messages

You now have a new method, but it is not yet parsing the `Host` field. You'll do this later on in the chapter, but before that, let's run the test function:

```
$ go test
--- FAIL: TestURLPort (0.00s)
    for host "foo.com:80"; got ""; want "80"
```

Wonderful! The method returned an empty string, and the test worked. Did you notice that you are using a different kind of error message this time? You used:

```
for host "foo.com:80"; got ""; want "80". want "80"
```

Instead of using:

```
&URL{Host: "foo.com:80"}.Port() = "";
```

I think the test name `TestURLPort` explains that the test is for the `Port` method. Also, the method does not take any arguments. So there is no need to include the type and the method in the error message. For example, in the Go Standard Library's official `url` package, they use the following message:

```
Port for Host %q = %q; want %q.
```

You may also have used something like:

```
"foo.com".Port()=%q; want %q".
```

As always, this is a matter of personal preference. So pick your poison and be consistent.

Why is the Port field a string and not a number?

In the Go Standard Library, ports are string values. This is because what appears after the colon can either be a numeric port number or a service name. For example, you can create a server listening on port 443 by typing: `net.Listen("tcp", "192.168.1.2:https")`. The port number for `https` is 443. You can also get the numeric port number for `https` by typing: `net.LookupPort("tcp", "https")`.

3.1.2 Maintainable test cases

It's time to get back to your primary goal: to test all the different variations of the `Host` field. As I explained, you have to test any input the Wizards team expects (Table 3.1), or they will cast a spell on you (they are wizards, after all).

Table 3.1. Tests for the Host field

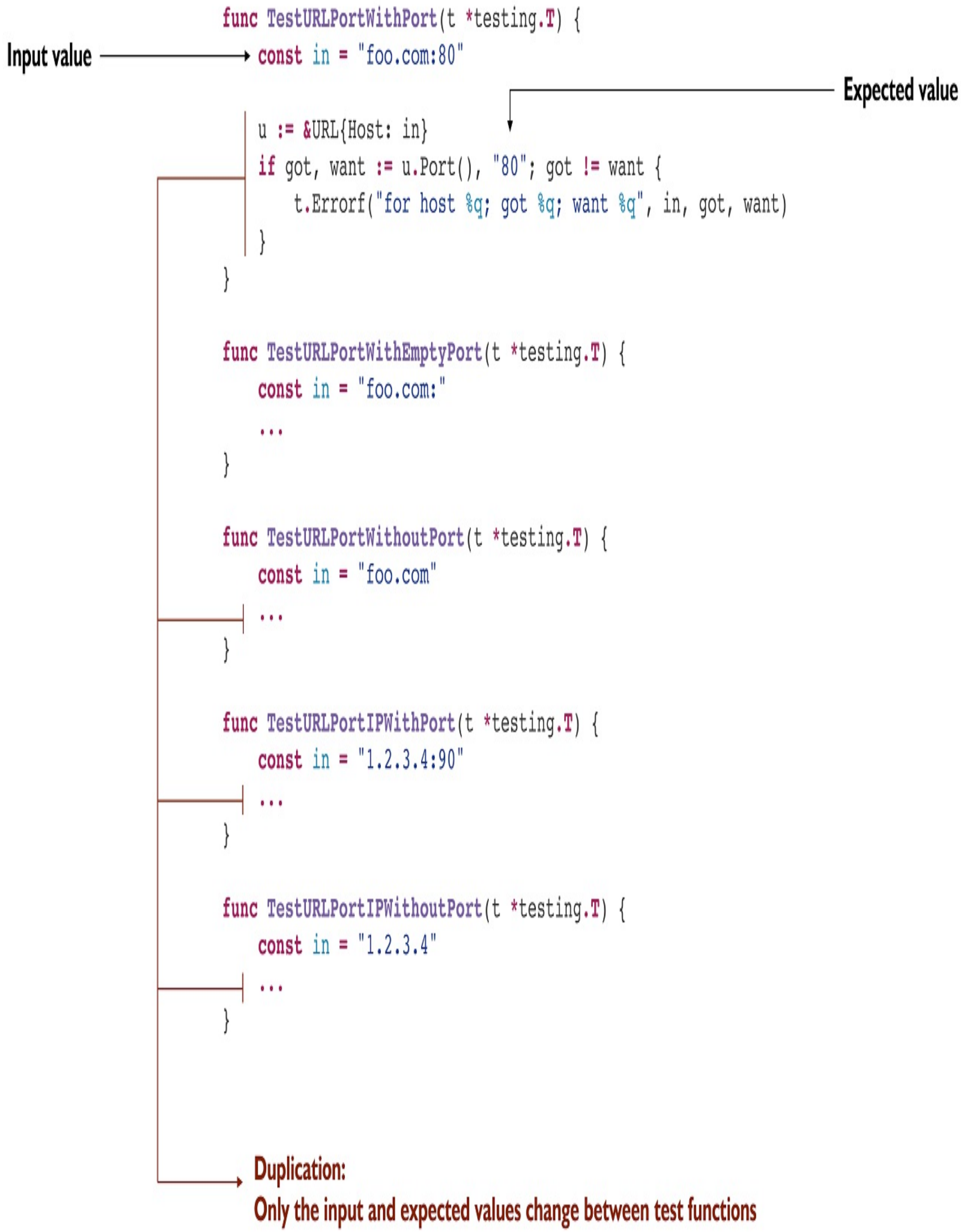
Test case	Host	Hostname	Port

With a port number	foo.com:80	foo.com	80
With an empty port number	foo.com:	foo.com	
Without a port number	foo.com	foo.com	
IP with a port number	1.2.3.4:90	1.2.3.4	90
IP without a port number	1.2.3.4	1.2.3.4	

Now you need to write new test functions for each test case in Table 3.1. But are you aware that only the inputs and expected port numbers are changing? Anyway, don't worry about this problem for now. I'll talk about it in a minute.

You will now create a test function and name it by looking at the "Test Case" column, provide the input value from the "Host" column, and check if the actual port number matches the number in the "Port" column. But, first, let's see the test functions you need to write in Figure 3.4.

Figure 3.4 Only the input host and expected port number values change between test functions. The rest of the code is similar to the previous test function in Listing 3.1. Writing tests this way creates unnecessary repetitive code.



When you run the tests, you should see something similar to the following:

```
$ go test
--- FAIL: TestURLPortWithPort
    for host "foo.com:80"; got ""; want "80"
--- FAIL: TestURLPortIPWithPort
    for host "1.2.3.4:90"; got ""; want "90"
```

Only two of the tests failed, and the rest passed. This is because the `Port` method returned an empty string, and the passing tests also expected an empty string. A match made in heaven! Joking aside, now there are some serious problems:

1. There's a lot of repetitive code without giving you any additional value. This is because all test functions share a similar code from Listing 3.1.
2. Also, if you need to test inputs you haven't guessed yet, you will need to add more duplicate tests in the future.
3. For example, the current parser does not support IPv6 addresses such as `[2001:db8:85a3:8d3:1319:8a2e:370:7348]:443`. So if you want to test IPv6 addresses, you will need to add more tests, resulting in more duplicate tests.

So, what's the solution? Two solutions come to my mind. Let's take a look at them:

Solution #1: Testing helpers

One approach is using a *test helper function* that only takes input data for the changing parts of the tests and bails out if something goes wrong. So the function will contain the same set of instructions as the previous one, except for the data it uses. It would be best if you also avoided the testing package to run this function. But, things will turn out okay as the function will take some input arguments and won't have a `Test` suffix. Let's take a look at that in Listing 3.2.

Listing 3.2: Writing an helper function (url_test.go)

```
func testPort(t *testing.T, in, wantPort string) {        #A
    u := &URL{Host: in}      #B
    if got := u.Port(); got != wantPort {       #C
        t.Errorf("for host %q; got %q; want %q", in, got, wantPor
```

```
    }
}
```

Note that the function takes a `*testing.T` parameter because it calls the `t.Errorf` method. As I showed you earlier, the `*testing.T` type is just a pointer to a struct, so you can pass it to other functions as you see here. The helper in Listing 3.2 takes these string parameters:

- `in`—The `Host` field that you want to test. The helper makes a new `URL` value for the input `Host` value to get the `Port` number.
- `wantPort`—The expected port number after parsing. The helper calls the `Port` method of the `URL` value and compares the result to that `port` value.

The helper becomes a reusable function with these two parameters. So test functions can call this function when testing for different `Host` values. You will implement a test function for each row in Table 3.1. Let's start with the first one in Listing 3.3.

Listing 3.3: Adding the first test case (url_test.go)

```
func TestURLPortWithPort(t *testing.T) {        #A
    testPort(t, "foo.com:80", "80")   #B
}
```

Note

A top-level test function starts with a `Test` prefix. For example, `TestURLPortWithPort` is a top-level test function. You'll soon learn that a test can contain subtests. A subtest is not a top-level test.

You might ask: "*Why is there a need for a test function that only calls another helper method?*" Note that the testing package only calls top-level test functions then generates a report for them. The `testPort` function is useless if you never call it. So it needs a test function. Win-win.

Let's run the test and see what it tells you:

```
$ go test
--- FAIL: TestURLPortWithPort (0.00s)
    url_test.go:5: for host "foo.com:80"; got ""; want "80"
```

Great! It works! Now roll up your sleeves and add the remaining test cases from Table 3.1 in Listing 3.4.

Tip

It's a convention to write single-line functions on the same line instead of adding the code after the curly braces. And as you can see in Listing 3.4, when you put them consecutively without empty lines between them, the `gofmt` tool will nicely align them.

Listing 3.4: Adding remaining test cases (url_test.go) func TestURLPortWithPort(t *testing.T) { testPort(t, "foo.com:80", "80") }

```
func TestURLPortWithEmptyPort(t *testing.T) { testPort(t, "foo.co
func TestURLPortWithoutPort(t *testing.T)   { testPort(t, "foo.co
func TestURLPortIPWithPort(t *testing.T)    { testPort(t, "1.2.3.
func TestURLPortIPWithoutPort(t *testing.T) { testPort(t, "1.2.3.
```

As you can see, each test function represents a row from Table 3.1 and each test for a different URL host. With these functions, you eliminate the duplication problem with one caveat. I hear you saying: "*What is it again? It never ends.*" Don't worry! It's a handy trick you'll be glad to learn (though I can't promise it!). Before that, let's run these tests and see what they tell you:

```
$ go test
--- FAIL: TestURLPortWithPort (0.00s)
    url_test.go:5: for host "foo.com:80"; got ""; want "80"
--- FAIL: TestURLPortIPWithPort (0.00s)
    url_test.go:5: for host "1.2.3.4:90"; got ""; want "90"
```

Did you notice the line numbers of the failure messages? They are all the same and make it harder for you to find which line the error has occurred. If you look at the earlier code in Listing 3.2, you can see that the `Errorf` call happens at line `5`. Is it a coincidence? Hardly. Let me give you a clue: The errors occurred in *the helper function* that all the testing functions share.

Ideally, you should expect to get exactly where an error occurs in a test function rather than in some random place. This isn't a big problem for these small tests but can be a serious issue for larger tests with lots of moving parts. Otherwise, you would need to work harder and waste your precious time to find the origin of the errors.

Can you somehow get the exact line number of the test function that fails instead of the helper function? Fortunately, there is a method called `Helper` on the `*testing.T` type that does that. You need to call it in the helper function as in Listing 3.5.

Listing 3.5: Marking the helper as a test helper (url_test.go)

```
func testPort(t *testing.T, in, wantPort string) {
    t.Helper()     #A
    ...
}
```

When you do this, the testing package will look at the function call stack and find the function that called this helper function. So, the caller, instead of the helper, will report the failure in the caller function. Let's take a look at the output:

```
$ go test
--- FAIL: TestURLPortWithPort (0.00s)
    url_test.go:1: for host "foo.com:80"; got ""; want "80"
--- FAIL: TestURLPortIPWithPort (0.00s)
    url_test.go:4: for host "1.2.3.4:90"; got ""; want "90"
```

When you look at the line numbers and compare them to Listing 3.4, you can see that the testing package now reports the correct test function where the error occurred.

Note

A test helper is not an actual test function but a helper for another test. However, it can still use the methods of the `*testing.T` type. Using `t.Helper()` in a test helper helps you find the actual location of an error. If you want to make a test fail, never return an error from a test helper, instead, use one of the `*testing.T` methods (as in Listing 3.2).

That's cool, but you may have guessed that writing and calling helper functions this way for each test can be cumbersome. And over time, they can pollute the namespace of your tests and increase complexity. So let's take a look at my second and best advice.

Solution #2: Table-driven testing

The idiomatic solution to the duplicate tests problem is *table-driven testing*. Table tests are all about finding repeatable patterns in your test code and identifying different use-case combinations. So, for example, instead of creating separate test functions to verify the same code with different inputs, you could put the *inputs* and *outputs* (*expectations*) in a table and write the test code in a *loop*.

The benefits of table-driven testing:

- Reduces the amount of repetitive test code you need to write
- Reduces cognitive load by having related test cases in a single test function
- Allows you to add new test cases in the future quickly
- Makes it easier to see if you've covered the corner cases
- Avoids adding helper methods for shared logic

First of all, you need to express the table in Table 3.1 in code using an anonymous struct:

```
struct {
    in   string // URL.Host field
    port string
}
```

Next, add the input data to a slice of the structs you just defined:

```
tests := []struct{...}{
    {in: "foo.com", port: ""},
    {in: "foo.com:80", port: "80"},
    // other inputs...
}
```

Let's take a look at the whole test function that tests the `Port` method using table-driven testing. As you can see in Listing 3.6, you can now write a test function that loops over the tests and verifies each input in a loop. As you can see, it is much easier to test the same logic with different inputs using table-driven tests.

Listing 3.6: Table-driven testing (url_test.go)

```
func TestURLPort(t *testing.T) {      #A
    tests := []struct {      #B
        in   string // URL.Host field #C
        port string          #D
    }{
        {in: "foo.com:80", port: "80"}, // with port      #E
        {in: "foo.com:", port: ""},     // with empty port
        {in: "foo.com", port: ""},      // without port   #E
        {in: "1.2.3.4:90", port: "90"}, // ip with port   #E
        {in: "1.2.3.4", port: ""},      // ip without port
        // Add more tests in case of a need     #F
    }
    for _, tt := range tests {        #G
        u := &URL{Host: tt.in}        #H
        if got, want := u.Port(), tt.port; got != want {  #I
            t.Errorf("for host %q; got %q; want %q", tt.in, got,
        }
    }
}
```

As you can see in Listing 3.6, the test code is almost identical to the previous code in Listing 3.2. The only difference is that the test code uses the input and output values from a table. First, you define an anonymous struct with the required fields:

- `in`—The `Host` field you want to test. The test creates a `URL` value for each `Host` value.
- `port`—The expected port after parsing. The test calls the `Port` method for each URL value and compares the result with that `port` value.

And you fill these fields using Table 3.1 for each host. After you create the `tests` slice, you loop over the slice and test the `Port` method using the values in the slice. The `tt` variable is a convention and avoids confusion with the `t` argument. You can think of it as a short version of saying a *table test*. Awesome! Now you have a table-driven test function, but there are still some problems with this approach.

3.1.3 Naming test cases

The table test you wrote in Listing 3.6 will fail when you run it, but at least it will give you descriptive error messages. But there are trade-offs everywhere. You reduced repetition by writing a table-driven test, but now you've lost something. If you compare the output of your table-driven test in Listing 3.6:

```
$ go test
--- FAIL: TestURLPort (0.00s)
    url_test.go:104: for host "foo.com:80"; got ""; want "80"
    url_test.go:104: for host "1.2.3.4:90"; got ""; want "90"
With the previous output of the tests that use a helper in Listin
--- FAIL: TestURLPortWithPort
    url_test.go:98: for host "foo.com:80"; got ""; want "80"
--- FAIL: TestURLPortIPWithPort
    url_test.go:101: for host "1.2.3.4:90"; got ""; want "90"
```

You'll notice that you have lost the test names. Previously, what you tested was clear with test function names: `TestURLPortWithPort` and `TestURLPortIPWithPort`. But they are not available in the table-driven test output now.

If you don't care about the test names yet, I'll tell you why they're such good friends. But before that, let's talk about another problem. Did you notice that you also lost the exact line numbers for the origin of the failures?

Since the test in Listing 3.4 used different test functions, the failure messages in its output showed different line numbers `98` and `101`. But in the table-driven test (Listing 3.6), both show the same line `104`. This is because the table-driven test uses the same code to verify different test cases. Line numbers are critical because they help you see and solve problems quickly.

You can't do anything for line numbers without using some hacky tricks without complicating the test. Fortunately, you can quickly find problematic test cases if you mark them somehow. One solution is to put numbers for each test case, as you can see in Listing 3.7.

Listing 3.7: Test cases with indexes (url_test.go)

```
func TestURLPort(t *testing.T) {
    tests := []struct {
        ...
    }{
```

```
        1: {in: "foo.com:80", port: "80"}  // with port   #A
        3: {in: "foo.com:", port: ""}      // with empty port
        2: {in: "foo.com", port: ""}       // without port
        4: {in: "1.2.3.4:90", port: "90"}  // ip with port
        5: {in: "1.2.3.4", port: ""}       // ip without port
    }
    for i := 1; i < len(tests); i++ { #B
        tt := tests[i]       #C
        ...
        t.Errorf("test %d: ...", i, ...)        #D
    }
}
```

Test functionality remained the same, except that each test case used a predefined index. For example, the 1st test case's input is `foo.com:80`, while the last one's input is `1.2.3.4`. You can pinpoint the failing test case fairly pretty quickly using indexes like this:

```
$ go test
--- FAIL: TestURLPort
    test 1: for host "foo.com:80"; got ""; want "80"
    test 4: for host "1.2.3.4:90"; got ""; want "90"
```

The first and the fourth tests failed. Besides showing descriptive failure messages, you can now quickly see which test cases failed. So you can go to the test function, find the failed test cases by their number and fix them.

Why is it important to manually add indexes?

You may already know that a slice type has index numbers for each element it contains. For example, let's say you have a slice:

```
myTestingStyle := []string{"table", "driven", "testing"}
```

You can get the elements using indexes like:

```
myTestingStyle[0] // table
myTestingStyle[1] // driven
myTestingStyle[2] // testing
```

What you may not know is that you can specify index numbers yourself. So, for example, you can make a slice as follows:

```
myTestingStyle := []string{1: "table", 2: "driven", 3: "testing"}
```

You can still retrieve elements using their index with this slice, but now they start at index `1` instead of `0`. For example, you can get the first element with `myTestingStyle[1]` instead of `myTestingStyle[0]`.

Why is this important? Imagine that some sets of test cases use 0-based indexes, where others use 1-based indexes. It wouldn't be obvious to other developers reading failure messages and, when a test case fails, they may be confused about which one failed. So it's better to be consistent.

Although, there won't be much problem if you're the only developer.

Getting rid of comments and numbers

You've written an excellent and clear table-driven test function with test cases with easy-to-find indexes. The sun is shining, there's a spring in your slippers, and you woke up with the scent of a delicious cup of coffee. Not so fast, though!

The problem with the previous test function is that you faked out the test names with indexes. I mean, can you see the comments next to each test case (Listing 3.7)? If you remember from the earliest tests you wrote, they were the names of the test functions. So now you hide them behind pesky numbers.

You might say: "*But you just said that indexes are better for finding failed test cases!*" Yes, I did so. I showed them because you can come across them on your testing journey in Go. So they will no longer shock you when you see them. Now I'm going to show you a better way of doing things in our modern era.

What about adding another field to the test table like this one:

```
tests := []struct {
    name string
    ...
}
```

With this, you can give each test case a name without using the comments that only appear in the test source code. So you can make them available in test failure messages as well. Win-win, right? Then you can write a test case like this:

```
{
    name: "with port",
    in:   "foo.com:80", port: "80",
}
```

Instead of this:

```
1: {in: "foo.com:80", port: "80"},  // with port
```

Anyway, let me show you the whole test function in Listing 3.8, so you can better understand what's going on.

Listing 3.8: Using named test cases (url_test.go)

```
func TestURLPort(t *testing.T) {
    tests := []struct {
        name string          #A
        in   string // URL.Host field
        port string
    }{
        {
            name: "with port",        #B
            in:   "foo.com:80", port: "80",
        },
        {
            name: "with empty port",  #B
            in:   "foo.com:", port: "",
        },
        {
            name: "without port",     #B
            in:   "foo.com", port: "",
        },
        {
            name: "ip with port",     #B
            in:   "1.2.3.4:90", port: "90",
        },
        {
            name: "ip without port",  #B
            in:   "1.2.3.4", port: "",
        },
    }
```

```
    for _, tt := range tests {
        u := &URL{Host: tt.in}
        if got, want := u.Port(), tt.port; got != want {
            t.Errorf("%s: for host %q; got %q; want %q", tt.name,
        }
    }
}
```

The table test in Listing 3.8 has a new field called `name`, and each test case in it has a name that describes its purpose. Besides that, when one of the test cases fails, it will give you more information about why it failed. So let's run the test and see what it looks like:

```
$ go test
--- FAIL: TestURLPort
    with port: for host "foo.com:80"; got ""; want "80"
    ip with port: for host "1.2.3.4:90"; got ""; want "90
```

As you can see, for example, when the `"with port"` test case fails, you can quickly search for it in the test function and see why. The test message tells you a few things like:

- The `"with port"` test has failed
- The test fails for the host value: `"foo.com:80"`
- The test wanted to get the port number `80`, but it got an empty string instead

Now you got rid of numbers and unnecessary comments. You may already know that comments are usually for those situations where code can't explain what it's doing. Thank goodness, your test cases and their messages are now self-explanatory.

I know the new way I proposed is more verbose but also more descriptive. Isn't that the whole point of writing tests in the first place? If you don't write descriptive test cases, then you're not writing well-mannered tests. I believe it is better to be descriptive than concise.

3.1.4 Wrap up

Congratulations! You now have a new tool in your arsenal to deal with complexity and reduce the cognitive load on your neural network. You know you need those neurons for better things. Let's see what you've learned so far:

- First, writing repetitive tests can be a real pain.
- You can use test helpers to mitigate the problem to some extent.
- If you have a set of inputs and outputs and the test code stays the same, you can use table-driven tests and make repetitive tests go out of the window.
- A table in a table test is usually a struct type where you define some fields for input and output values and validate against them in a loop.
- It's vital to pinpoint the source of a failure message. If you don't name your test cases, you may miss the origin of a failure when using table-driven testing.

3.2 Subtests

A subtest is a standalone test similar to a top-level test. Subtests allows you to run a test under a top-level test in isolation and choose which ones to run. Subtest is a test that you can run within a top-level test function in isolation.

- Isolation allows a subtest to fail and others to continue (even if one of them fails with `Fatal` and `Fatalf`).
- It also allows you to run each subtest in parallel if wanted.
- One more advantage is that you can choose which subtests to run.

On the other hand, table-driven testing can help you up to a point. You run the tests using a data table under the same top-level test, and these tests are not isolated from each other. For example, if you ever use the `Fatal` or `Fatalf` functions, the whole test function will stop running and fail without running the rest of the tests in the table. There is only a single test in a table test: the top-level test function that loops over a data table and makes assertions.

However, table-driven testing is still valuable. Especially, when you combine it with subtests. I'll show you how to do so later in this chapter. Before doing that, it's time to discuss the isolation problem in detail.

3.2.1 Isolation problem

You are proud that you have good table-driven test cases. You ran the tests once again and found that you still have two failing test cases:

```
$ go test
--- FAIL: TestURLPort
    with port: for host "foo.com:80"; got ""; want "80"
    ip with port: for host "1.2.3.4:90"; got ""; want "90
```

But you want to focus on the `"with port"` test case and fix it without running the other test cases. Table-driven tests are useful enough in most cases, but the problem here is that you can't debug each failed test case individually to figure out the underlying problem. So how can you run a particular test case? Let's discuss what you can do about it.

Solution: Commenting out

Since you are using a table-driven test, you cannot run only the first test that fails without running the others. This is because each test case belongs to a single test function (`TestURLPort`), and they all run in tandem.

If you're feeling adventurous, as a quick fix, you could comment out all the test cases but the one you want to work on. By doing that, you would only see the first test case when you run the tests.

Do you think this would be a good solution? Sometimes, maybe. Currently, you have relatively few test cases. For rare cases, quick hacks might be fine, but if you had dozens of (or more) test cases, it would not be easy to comment out every one of them and understand what's going on.

Solution: Fatalf

So, is there a convenient way to just run the first failed test without doing any hackery magic? For example, you could use the `Fatalf` method from chapter 2 instead of the `Errorf` method to work on the first failed test. But there are still some problems here.

What if you want to work on the second failed test case (`"ip with port"`) instead of the first one? In that case, you would either use the comment-out hack, or you wouldn't use a table-driven test at all.

On top of that, using the `Fatalf` method won't stop the other test functions. Also, changing the test code can be dangerous if you make a mistake. You might want to use our old friend `Fatalf` method if a critical error happens and makes running the rest of the test function pointless. But I don't think mismatching port numbers is that deadly. On the other hand, you may want to see all other failed test cases to get an overall picture of the code and better guess what went wrong.

What if tests are shuffled?

You should never write tests that depend on another so you can run them in any order. Otherwise, no matter how careful you are, there will always be a possibility of false results. And you may end up working hard to figure out which tests are causing the real failure.

Before discussing how to run tests in random order, let me talk about the `verbose` flag. As you know, the testing package is relatively quiet unless you have a failed test. But when you use the verbose flag, the testing package becomes chatty and shows you all the tests it has run. This is handy when you shuffle your tests. I'll be back to this very soon. But, first, let's look at how your tests' output changes when you use the verbose flag:

```
$ go test –v
=== RUN   TestParse
--- PASS: TestParse
=== RUN   TestURLString
--- PASS: TestURLString
=== RUN   TestURLPort
    ...
--- FAIL: TestURLPort
...
```

Alright, it's time to talk about how to run these tests in random order. As of Go 1.17, a flag called `shuffle` allows you to shuffle the execution order of tests. So each time you run the tests, the testing package will run them in a different order:

```
$ go test -v -shuffle=on
-test.shuffle 1624629565232133000
=== RUN   TestURLString
--- PASS: TestURLString
=== RUN   TestURLPort
...
--- FAIL: TestURLPort
...
$ go test -v -shuffle=on
=== RUN   TestURLPort
...
--- FAIL: TestURLPort
=== RUN   TestParse
--- PASS: TestParse
...
```

If you take a closer look at the output, you can see a number:

```
1624629565232133000
```

When some tests fail due to shuffling, you can run the tests in the same order by giving that number to the shuffle flag:

```
$ go test -v -shuffle=1624629565232133000
```

By the way, shuffling only works for top-level test functions and does not shuffle test cases in a table-driven test. Fortunately, I know a trick that lets you do that: Ranging over a map returns its elements in random order. So when defining a test table, you can use a *map type* instead of a struct type (Listing 3.9).

Note

The iteration order is random because people started depending on the iteration order of maps in the early days of Go. So code that worked on one machine did not work on another. If you're curious, you might want to read this: https://golang.org/doc/go1#iteration.

Listing 3.9: Shuffling test cases with a map (url_test.go)

```
func TestURLPort(t *testing.T) {
    tests := map[string]struct {      #A
#B
        in   string // URL.Host field
        port string
    }{
        "with port":       {in: "foo.com:80", port: "80"},
        "with empty port": {in: "foo.com", port: ""},     #C
        "without port":    {in: "foo.com:", port: ""},    #C
        "ip with port":    {in: "1.2.3.4:90", port: "90"},
        "ip without port": {in: "1.2.3.4", port: ""},     #C
    }
    for name, tt := range tests {     #D
        ...
        if ...; got != want {
            t.Errorf("...", name, ...) #E
        }
    }
}
```

- `map[string]`—The map keys that describe the test case names.
- `struct{...}`—The map elements describe the inputs and expected values. You no longer need to add a name field to the `struct` because the map keys already describe the test case names.

In Listing 3.9, the test table uses a map type instead of a struct type. Earlier, you wrote a table-driven test using a struct type, but it doesn't always have to be this way. Although we gophers often use structs, this is not a rule, and you can be creative—It's a standard test function, after all.

It's better to take advantage of the randomized nature of a map so that you can quickly find possible dependency issues in tests and code. So, every time you run this test, Go will randomly select the test cases from the `tests` table. Let's take a look at the output:

```
$ go test -v -shuffle=on
=== RUN   TestURLString
--- PASS: TestURLString
=== RUN   TestURLPort
    with port: for host "foo.com:80"; got ""; want "80"
    ip with port: for host "1.2.3.4:90"; got ""; want "90
--- FAIL: TestURLPort
$ go test -v -shuffle=on
=== RUN   TestURLPort
```

```
    ip with port: for host "1.2.3.4:90"; got ""; want "90
    with port: for host "foo.com:80"; got ""; want "80"
--- FAIL: TestURLPort
=== RUN   TestURLString
--- PASS: TestURLString
```

Warning

You still need to use `` `shuffle=on` ``. Otherwise, the test package won't run top-level tests in random order, such as `TestURLString` and `TestURLPort`. Remember, the shuffle option only affects the top-level tests.

Now getting the first failed test has become even more complicated. No worries. Time to talk about the solution: *subtests*.

3.2.2 Writing your first subtest

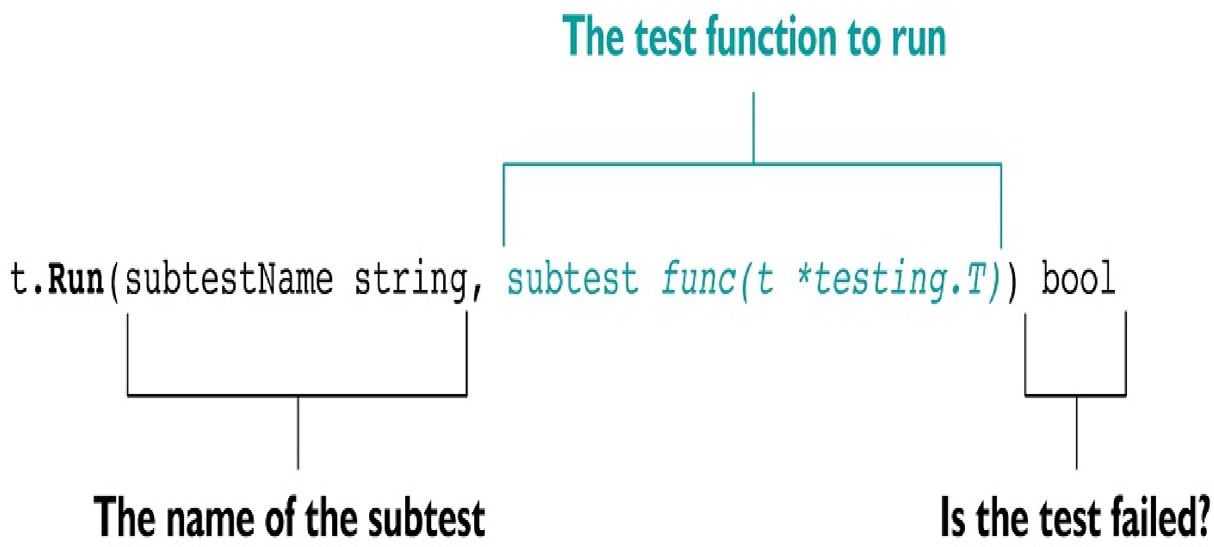
You want to run a specific test case to focus on, but you can't because it's in a table-driven test. What if I told you that you could hit two drones with one stone? If you had run the test cases as *subtests*, you could choose which one to work on.

But before writing your first subtest, let's talk about what makes a test a subtest. After that, you'll be ready to write your first subtest without using table-driven tests. This way, you can better see how a subtest works. After that, you will learn how to combine subtests with table-driven tests. Let's get started!

What makes a test a subtest?

Let's first see how to run a subtest and then discuss what makes a test a subtest. You can do so by calling the `Run` method of the `*testing.T` type within a top-level test function (Figure 3.5).

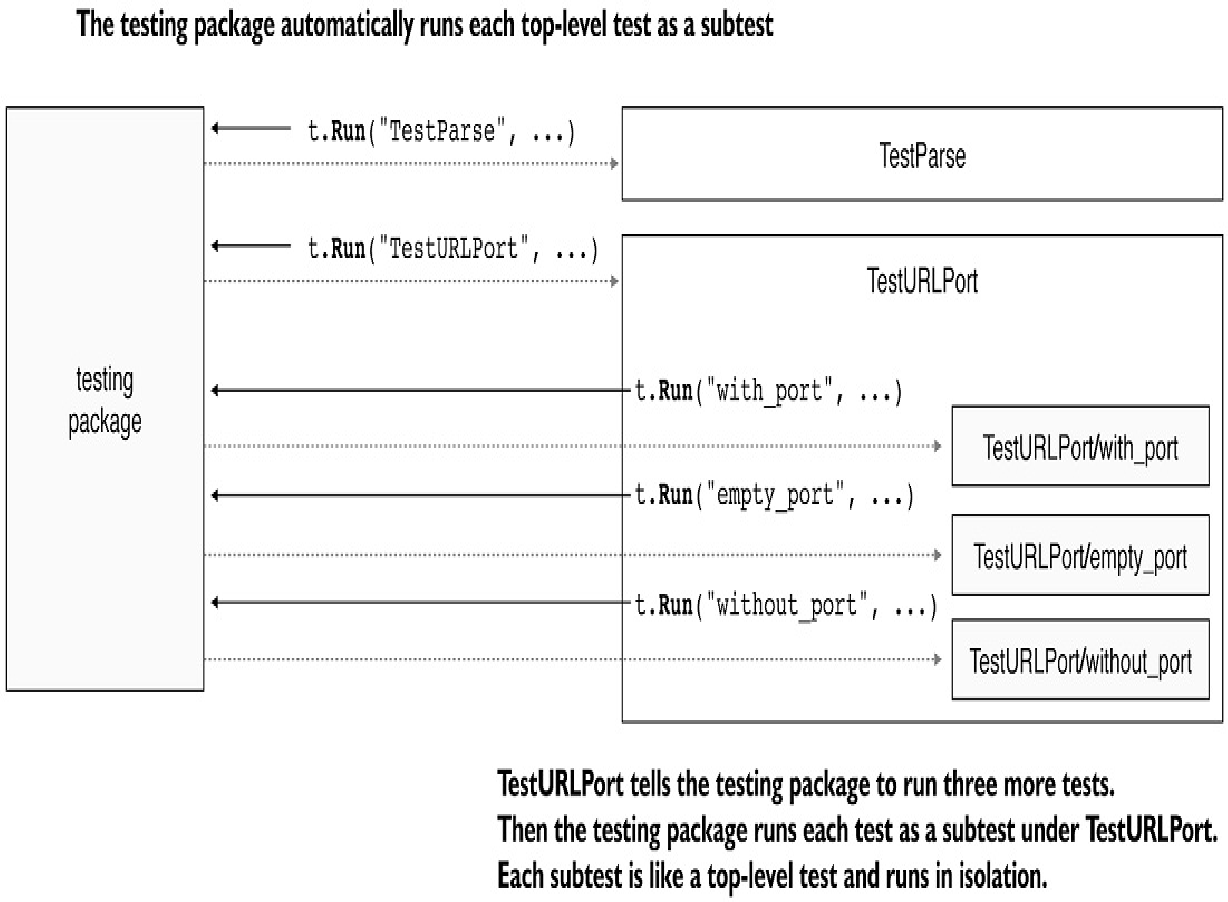
Figure 3.5 The t.Run method can run a subtest



- Each subtest has a name. When the testing package runs a top-level test function, it automatically uses a test function's name. However, when you want to run a subtest yourself, you might want to give it a name, or the testing package will pick a unique number as its name.
- The testing package will run the function using the `subtest` input value. You may have already guessed that the signature of this function is the same as a top-level test function.

Since you understand how to run a subtest, let's discuss what happens when you run them under a top-level test. In Figure 3.6, the testing package automatically runs two top-level test functions by itself, and the last one, `TestURLPort`, tells the testing package to run three more tests as subtests.

Figure 3.6 The testing package runs each test function as a subtest. And test functions can also run their own subtests.



As you can see in Figure 3.6, there's something I didn't tell you before: From the moment you wrote your first test, you were using subtests! This is because the testing package calls each top-level test function as a *subtest* under the hood. As of Go 1.7, the testing package exposed the same functionality to gophers.

- There are two top-level test functions in Figure 3.6: `TestParse` and `TestURLPort`.
- The testing package can automatically run the top-level tests as subtests. Behind the scenes, it calls the `t.Run` method to run each top-level test.
- The second top-level test wants to run three subtests. So, it needs to tell the testing package to do so. The testing package will run these tests as subtests under the `TestURLPort` test. Each subtest will run in isolation, just like a top-level test. A fatal test failure in a subtest won't affect the others.

As explained, each top-level test is also a subtest. And a top-level function can also run its subtests. The main advantage of using subtests is to run them in isolation and have more granular control over your tests. You'll better understand what I mean in the next sections.

Testing with subtests

Let's write a test function from scratch so I can show you subtests from the ground up. First, let's delete the code in the `TestURLPort` function and then write each test case as a subtest in Listing 3.10.

Listing 3.10: A subtest within a test function (url_test.go)

```
func TestURLPort(t *testing.T) {      #A
    t.Run("with port", func(t *testing.T) {     #B
        const in = "foo.com:80"       #C

        u := &URL{Host: in}  #C
        if got, want := u.Port(), "80"; got != want {     #C
            t.Errorf("for host %q; got %q; want %q", in, got, wan
        } #C
    })  #B
    t.Run("empty port", func(t *testing.T)      { /* similar code
    t.Run("without port", func(t *testing.T)    { /* similar code
    t.Run("ip with port", func(t *testing.T)    {         #C
        const in = "1.2.3.4:90"       #C

        u := &URL{Host: in}  #C
        if got, want := u.Port(), "90"; got != want {     #C
            t.Errorf("for host %q; got %q; want %q", in, got, wan
        } #C
    })  #C
    t.Run("ip without port", func(t *testing.T) { /* similar code
}
```

As you can see, there is a duplication problem in subtests. I'll talk about the duplication problem later, but for now, let's focus on the first subtest:

- In line 2, you pass the name of the test case as `"with port"` via the first parameter of the `Run` method. The test package is now aware that a subtest named `"with port"` is in the `TestURLPort` top-level test.
- Then you pass an anonymous test function as a subtest to the `Run`

method. So the testing package will run this subtest when it runs the `TestURLPort` function.

By the way, keep the line numbers in mind (Listing 3.10):

- The "`with port`" subtest is in line 2.
- The "`ip with port`" subtest is in line 12.

It's time to run the test and see what its output looks like in Listing 3.11.

Listing 3.11: The "with port" subtest's output

```
--- FAIL: TestURLPort        #A
    --- FAIL: TestURLPort/with_port   #B
        url_test.go:2: for host "foo.com:80"; got ""; want "80"
    --- FAIL: TestURLPort/ip_with_port #B
        url_test.go:12: for host "1.2.3.4:90"; got ""; want "90"
```

- The testing package prints subtests hierarchically under their parent test function. This makes it easier to see where a subtest belongs.
- The parent test of a subtest also fails when one of their subtests fails because they work as a group. But the parent test continues to run the other subtests. Just like the way it happens with top-level tests.

Did you notice that each subtest reported different line numbers? Let's take a look at the previous code in Listing 3.10. The first failure occurred on line `2`, and there you can see that the first subtest is calling the `Errorf` method. This is the origin of the first error. You can also find the origin of the subsequent failure by looking at line `12`. Since you're now using subtests, you can quickly pinpoint the source of the failures.

The failfast flag

In the previous section, you wanted to work on the first failing test. So you can focus and fix it without dealing with other tests. But, unfortunately, you haven't been able to find a suitable solution for this problem so far.

As of Go 1.10, you can use the `failfast` flag, which the lovely author of yours proudly contributed to the Go testing package. It's for stopping tests in a single package if one of them fails. For example, let's say you want to work on the first failing test. Then you can use it as follows:

```
$ go test –failfast
--- FAIL: TestURLPort
    --- FAIL: TestURLPort/with_port (0.00s)
    url_test.go:2: for host "foo.com:80"; got ""; want "80"
```

Cool, isn't it? It's that simple. But if you had used table-driven test cases without subtests, the `failfast` flag would not have stopped other test cases from running. This is because table-driven test cases are not subtests, and the flag only stops subtests (*remember: top-level tests are also subtests*).

The run flag

Sun is shining, and the grass is greener, but now it's raining. What if you wanted to run a specific test instead of the first one that failed? You can't do that with the `failfast` flag. But you can do it using the superpowers of the subtests. Before explaining how to do that, let's look at how you can run a specific subtest using the `run` flag. As you might know now, top-level tests are also subtests. So you can also use the `run` flag to run them:

```
$ go test -v -run=TestParse
=== RUN   TestParse
--- PASS: TestParse
```

What if there were other test functions that their names start with `TestParse`? For example, let's say there is another test called `TestParseScheme`. The `run` flag runs both:

```
$ go test -v -run=TestParse
=== RUN   TestParse
=== RUN   TestParseScheme
...
```

As you can see, the `run` flag ran two tests that contained `TestParse` in their name: `TestParse` and `TestParseScheme`. But how can you run `TestParse` only? If you know a thing or two about *regular expressions*, then you're home because the `run` flag takes regular expressions to find your tests:

```
$ go test -v -run=TestParse$
=== RUN   TestParse
...
The dollar sign ($) means the end of a line. So the run flag only
$ go test -v -run=TestURLPort$
=== RUN   TestURLPort
=== RUN   TestURLPort/with_port
    for host "foo.com:80"; got ""; want "80"
=== RUN   TestURLPort/with_empty_port
=== RUN   TestURLPort/without_port
=== RUN   TestURLPort/ip_with_port
    for host "1.2.3.4:90"; got ""; want "90"
=== RUN   TestURLPort/ip_without_port
```

So what's the solution? With the *slash character (/)*, you can select subtests within subtests. There are no limitations on the levels of the subtests. In practice, we gophers often use two to three levels at most. For example, if you want to work on the `with_port` subtest of the `TestURLPort` test, you can do it as follows:

```
$ go test -run=TestURLPort/^with_port
--- FAIL: TestURLPort (0.00s)
    --- FAIL: TestURLPort/with_port (0.00s)
    url_test.go:2: for host "foo.com:80"; got ""; want "80"
```

Or let's say you want to work on the `ip_with_port` subtest. You can do it like this:

```
$ go test -run=TestURLPort/ip_with_port
--- FAIL: TestURLPort (0.00s)
    --- FAIL: TestURLPort/ip_with_port
    url_test.go:12: for host "1.2.3.4:90"; got ""; want "90"
```

Wondering why you use a caret sign *(^)* in the first command but not on the other? A caret sign means the beginning of a line. If you run the first command without it, you'll get both subtests because each contains `with_port` in its name:

```
$ go test -run=TestURLPort/with_port
--- FAIL: TestURLPort (0.00s)
    --- FAIL: TestURLPort/with_port ...
    --- FAIL: TestURLPort/ip_with_port ...
```

It would be unfair to finish this topic here without explaining one more interesting aspect of the `run` flag. The testing package separates the value you give the `run` flag with slashes and interprets each as *separate regular expressions*.

So, for example, let's say there is another test called `TestURLString`. You can find these top-level tests with "`URL`" *and* subtests with "`without`" with the following command:

```
$ go test -v -run=URL/without
=== RUN   TestURLString
=== RUN   TestURLPort
=== RUN   TestURLPort/without_port
=== RUN   TestURLPort/ip_without_port
```

- The `run` flag runs `TestURLString` and `TestURLPort` because each one has "`URL`" in its name.
- But it only runs the subtests of `TestURLPort` that contain "`without`" in their name.
- If there were another test function like: `"TestDetect/without_port"`, it wouldn't run it because `TestDetect` doesn't contain "`URL`" in its name even though it had a "`without_port`" subtest.

You can naturally understand how the `run` flag acts and reacts; try running your tests with it. Play with it! You now have superpowers to run any test you want. Well done!

Note

Regular expressions are beyond the scope of this book. If you're curious, Go uses the RE2 regular expression syntax. You can learn more about it here: https://github.com/google/re2/wiki/Syntax.

3.2.3 Avoiding duplication

If you're like me, you find the code in Listing 3.10 unnecessarily lengthy and repetitive. You're trying to understand why everything you've learned so far is necessary. You're back to square one, are you? Don't worry. Things you learned so far are fantastic, and you can use them to your advantage. In this section, I will show you some possible solutions.

Combining subtests with a test helper

Thank goodness, you know everything about test helpers. So let's move the repetitive logic in subtests to a test helper (just like you did in section 3.1.2). Remember, when you use a test helper, you can get the exact line number of a failure if it happens.

Listing 3.12: Moving the shared logic to a test helper (url_test.go)

```
func TestURLPort(t *testing.T) {
    testPort := func(in, wantPort string) {     #A
        t.Helper() #B
        u := &URL{Host: in}  #C
        if got := u.Port(); got != wantPort {   #C
            t.Errorf("for host %q; got %q; want %q", in, got, wan
        } #C
    }   #A
    t.Run("with port", func(t *testing.T) { testPort("foo.com:80"
    t.Run("with empty port", func(t *testing.T) { testPort("foo.c
    t.Run("without port", func(t *testing.T) { testPort("foo.com"
    t.Run("ip with port", func(t *testing.T) { testPort("1.2.3.4:
    t.Run("ip without port", func(t *testing.T) { testPort("1.2.3
}
```

The `testPort` test helper is an anonymous function that is only visible in the `TestURLPort` test function. So it doesn't pollute the namespace of your tests. It takes a `Host` value as input and an expected port number. Then it verifies the expected port number matches the port number of the host. And if it's not, it fails the test. Finally, each subtest calls the test helper you wrote with its custom inputs and expected values.

This can be a good solution for a small number of tests such as this one in Listing 3.12. But it can quickly get out of control for a large number of tests. So use it with caution.

Using a higher-order function

Each subtest in Listing 3.12 wraps the test helper in a test function like this:

```
t.Run("name here", func(t *testing.T) {
    testPort("host here", "port here")
})
```

But if you had used a higher-order function, you could have used it like this:

```
t.Run("name here", testPort("host here", "port here"))
```

Let's take a look at Listing 3.13 if you're wondering how you can do so.

Listing 3.13: Using a higher-order function (url_test.go)

```
func TestURLPort(t *testing.T) {
    testPort := func(in, wantPort string) func(*testing.T) {
        return func(t *testing.T) {   #A
            t.Helper()
            u := &URL{Host: in}
            if got := u.Port(); got != wantPort {
                t.Errorf("for host %q; got %q; want %q", in, got,
            }
        } #A
    }
    t.Run("with port", testPort("foo.com:80", "80"))       #B
    t.Run("with empty port", testPort("foo.com:", ""))    #B
    t.Run("without port", testPort("foo.com", ""))        #B
    t.Run("ip with port", testPort("1.2.3.4:90", "90"))   #B
    t.Run("ip without port", testPort("1.2.3.4", ""))     #B
}
```

A tad drier, isn't it? The `testPort` function is now returning an anonymous test helper function: `func(*testing.T)`. In Go, functions are first-class citizens so that you can return a function from another function. When you do this, the function that returns another function is called a *higher-order function*. The `Run` method expects a test function: A function that takes a `*testing.T` parameter. Since the test helper already returns a test function, you can pass it to the `Run` method. This is a rather bizarre way of doing testing, but sometimes you may see it in the wild. But I think it's unnecessarily complex.

Combining subtests with table-tests

Let's take a look at the idiomatic way of testing with subtests. When you combine table-driven tests with subtests, you can have all the benefits of table-driven tests and subtests such as:

- Ability to run subtests in isolation
- Keeping the code concise by taking advantage of subtests

I'll talk about the last one soon. But before that, let me show you how to combine a table-test with a subtest in Listing 3.14.

Listing 3.14: Combining a table-test with subtests (url_test.go)

```
func TestURLPort(t *testing.T) {
    tests := map[string]struct {      #A
        in   string // URL.Host field
        port string
    }{
        "with port":       {in: "foo.com:80", port: "80"},
        ...        #B
    }
    for name, tt := range tests {
        t.Run(name, func(t *testing.T) {        #C
            u := &URL{Host: tt.in}
            if got, want := u.Port(), tt.port; got != want {
                t.Errorf("for host %q; got %q; want %q", tt.in, g
            }
        })
    }
}
```

Since you can programmatically run a subtest, you can run it in a table-driven test as well. First, you define a table test and put your test cases in it. Then, you loop over the test cases and run each as a subtest using the `Run` method. Now you have all the benefits of table-driven tests plus subtests! With the addition of subtests, you now have the ability to use the `run` flag to run a specific set of subtests.

Let's run the tests and see the output:

```
$ go test
--- FAIL: TestURLPort
    --- FAIL: TestURLPort/with_empty_port
        for host "foo.com:80"; got ""; want "80"
    --- FAIL: TestURLPort/ip_with_port
```

```
        for host "1.2.3.4:90"; got ""; want "90"
```

The failure messages are descriptive enough, but they are a bit lengthy. They print out the host value every time they fail. But when you write a large number of tests, you always want to read less. Let's see how you can make them better next.

Making the failure messages concise

As you know, you can give any name you want to a subtest. So why don't you take this power to your advantage? For example, you could do something like this:

```
$ go test
--- FAIL: TestURLPort
    --- FAIL: TestURLPort/with_empty_port/foo.com:80
        got ""; want "80"
    --- FAIL: TestURLPort/ip_with_port/1.2.3.4:90
        got ""; want "90"
```

Now the host values look like subtests, but actually, they are not, and they don't have to be. You can be creative! Here, the host values are now part of the subtests hierarchy for reducing complexity:

- `TestURLPort` is a top-level test function.
- `with_empty_port` is a subtest of the `TestURLPort`.
- `foo.com:80` is a subtest of the `with_empty_port`.

Not only did this make the output clearer, but it also enabled something else. You can now run any subtests using their name to verify a specific host value. For example, let's say you want to see only the subtests that verify `foo.com`:

```
$ go test -v -run=TestURLPort//foo.com
=== RUN   TestURLPort
=== RUN   TestURLPort/with_port/foo.com
=== RUN   TestURLPort/with_empty_port/foo.com:80
    url_test.go:330: got ""; want "80"
=== RUN   TestURLPort/without_port/foo.com:
```

The double-slashes (`//`) are necessary because you're now matching a subtest within another subtest. For example, the current selector first matches to the top-level `TestURLPort` test, then it matches any subtests within it, and finally, it picks subtests that contain "`foo.com`."

Tip

The testing package separates tests by slashes. You can use the double-slashes as a shortcut instead of typing `TestURLPort/*/foo.com`.

I know there are no real subtests for host values, but that doesn't change anything here. This is because the `run` flag operates on strings separated by slashes. For example:

- The `run` flag will see three subtests in `TestURLPort/with_port/foo.com`
- `TestURLPort`, `with_port`, and `foo.com`.
- In reality, there is a top-level test function named `TestURLPort`, and a subtest named `with_port`.
- So there are actually two levels of subtests, but the `run` flag evaluates it as three levels because it matches subtests by their name separated by slashes.

Now you've seen what you wanted to achieve, let's realize this in the code (Listing 3.15).

Listing 3.15: Implementing better failure messages (url_test.go)

```
func TestURLPort(t *testing.T) {
    tests := map[string]struct {
        in   string // URL.Host field
        port string
    }{
        "with port":       {in: "foo.com:80", port: "80"},
        // ...other tests
    }
    for name, tt := range tests {
        t.Run(fmt.Sprintf("%s/%s", name, tt.in), func(t *testing.
            u := &URL{Host: tt.in}
            if got, want := u.Port(), tt.port; got != want {
```

```
                t.Errorf("got %q; want %q", got, want)    #B
            }
        })
    }
}
```

- Since `TestURLPort` is a top-level test function and automatically gets a test name in the test results, you don't have to give it a name.
- In `"%s/%s"`, the first `%s` is the name of one of the test cases in the tests table. And the second one is a host value that you get from the tests table.
- Here, the `Sprintf` function is unnecessary but makes it easier to extend the test name in the future. You can simply combine these string values yourself if you don't like this style!

About fatal failures

Let's say you're using a table-driven test, and one of the test cases fails with methods like `Fatal`, `Fatalf`, `FailNow`. So, the testing package won't run the remaining test cases. But sometimes, you want to see the overall picture of the code you're testing.

Thank goodness, there are no such problems with the subtests, as they are great! Jokes aside, since each subtest is a test in itself, if it fails, it won't affect the other tests, and they won't jump into the abyss together. So you can safely crash a subtest, and the other tests will keep going on. This reduces the cognitive load on your part, so you can forget about affecting other tests if one of them fails.

3.2.4 Wrap up

Wow, what a journey! You started with a simple problem and wanted to run a test in isolation but look at how many things you've learned so far:

- Subtests are test functions that can be programmatically called.
- Top-level test functions are also subtests under the hood.
- Subtests allow running table-driven test cases in isolation.
- Subtests make failure messages concise and descriptive.

- Subtests help to organize tests in a hierarchy.

This was just the beginning, as there is more to learn about subtests like grouping parallel tests, managing setup, and teardown stages. You'll learn them later in the book. But, for now, this was a great start.

3.3 Implementing the parser

Let's remember what the Wizards team was asking. They recently realized that they were receiving URLs with port numbers, and they wanted to deny access to some of the hostnames and ports. So far, you learned about testing with table-driven tests and subtests. The time has come to implement the `Hostname` and `Port` methods for real and make the tests finally pass according to Table 3.2. Let's get started!

Table 3.2. Shared test cases both for the Port and Hostname methods

Test case	Host	Hostname	Port
With a port number	foo.com:80	foo.com	80
With an empty port number	foo.com:	foo.com	
Without a port number	foo.com	foo.com	
IP with a port number	1.2.3.4:90	1.2.3.4	90
IP without a port number	1.2.3.4	1.2.3.4	

Testing the Hostname method

So far, you've only written tests for the `Port` method, not `Hostname`. However, these methods go in tandem as the `Port` method parses the port number part of a `Host` value while the `Hostname` parses the hostname part. So you can use the same test cases in Table 3.2 for the `Hostname` method. Now you've got to decide! You have a test function called `TestURLPort`.

1. Are you going to create a new test function called `TestURLHostname` for the `Hostname` method?
2. Or, are you going to change the `TestURLPort` test function's name to `TestURLHost` and add one more subtest for the `Hostname` method?

Let's start with the first approach. To share the same test cases with the `TestURLPort` function, you can move the test cases as a package-level variable and share it across the test functions (Listing 3.16).

Listing 3.16: Sharing the test cases (url_test.go)

```
var hostTests = map[string]struct {   #A
    in       string // URL.Host field
    hostname string          #B
    port     string
}{
    "with port":       {in: "foo.com:80", hostname: "foo.com", po
    "with empty port": {in: "foo.com", hostname: "foo.com", port:
    "without port":    {in: "foo.com:", hostname: "foo.com", port
    "ip with port":    {in: "1.2.3.4:90", hostname: "1.2.3.4", po
    "ip without port": {in: "1.2.3.4", hostname: "1.2.3.4", port:
}
```

You added a new field called hostname and provided the expected hostname values for the `TestURLHostname` test function. So you can check whether the `Hostname` method correctly parses the hostname from the `Host` field. Let's use the shared test cases in both test functions in Listing 3.17.

Listing 3.17: Using the shared test cases (url_test.go)

```
func TestURLHostname(t *testing.T) {  #A
    for name, tt := range hostTests { #B
        t.Run(fmt.Sprintf("%s/%s", name, tt.in), func(t *testing.
```

```
            u := &URL{Host: tt.in}
            if got, want := u.Hostname(), tt.hostname; got != wan
                t.Errorf("got %q; want %q", got, want)
            }
        })
    }
}

func TestURLPort(t *testing.T) {
    for name, tt := range hostTests { #B
        t.Run(fmt.Sprintf("%s/%s", name, tt.in), func(t *testing.
            u := &URL{Host: tt.in}
            if got, want := u.Port(), tt.port; got != want {
                t.Errorf("got %q; want %q", got, want)
            }
        })
    }
}
```

Before running the tests, for now, let's implement the `Hostname` method that returns an empty string:

```
func (u *URL) Hostname() string {
    return ""
}
```

The output will look similar to the following if you run the tests:

```
$ go test
--- FAIL: TestURLHostname
    --- FAIL: TestURLHostname/ip_with_port/1.2.3.4:90
        got ""; want "1.2.3.4"
    --- FAIL: ...
--- FAIL: TestURLPort
    --- FAIL: TestURLPort/ip_with_port/1.2.3.4:90
        got ""; want "90"
    --- FAIL: ...
```

As you can see, now both test functions run the same test cases because they share the same test table called `hostTests`. The first test function tests the `Hostname` method, and the other tests the `Port` method. So both test functions do very similar work.

I feel comfortable testing this way because each test function clearly describes what methods they're testing. But you can still use the subtest-superpowers and run them hierarchically in the same test function without losing anything. So let's now try the second approach (Listing 3.18).

Listing 3.18: Sharing the test cases in the same function (url_test.go)

```
func TestURLHost(t *testing.T) {
    tests := ...   #A
    for name, tt := range tests {
        t.Run(fmt.Sprintf("Hostname/%s/%s", name, tt.in), func(t
            ...
        })
        t.Run(fmt.Sprintf("Port/%s/%s", name, tt.in), func(t *tes
            ...
        })
    }
}
```

Now you have a single test that tests both for the `Hostname` and `Port` methods. The first `Run` call is for the `Hostname` subtests, and the second is for the `Port` subtests. Each one groups their subtests by adding a prefix to their names for what they're testing. For example, the first group of subtests starts with a `"Hostname/"` prefix.

So you can now run subtests in a granular fashion. For example, you can run all the subtests with this:

```
$ go test -v -run=TestURLHost
```

Or, you can run only the `Hostname` subtests:

```
$ go test -v -run=TestURLHost/Hostname
```

Or, you can run the `Port` subtests that test only hostnames with port:

```
$ go test -v -run=TestURLHost/Port/with_port
```

The test function's name is somewhat unusual: `TestURLHost`. You usually want to name your test functions for the behavior they test. So the prior approach was better in that sense. But the current approach is concise and doesn't pollute the test namespace, so I like the current one more. Since the

`Hostname` and `Port` methods are for parsing the `Host` field of the `URL` type, I think it's okay to name this test function as `TestURLHost`.

Implementing the methods

You have tests that fail because you didn't implement the `Hostname` and `Port` methods yet. Now you're ready to implement the methods for real. Let's start with the `Hostname` method. It will parse the hostname from the `Host` field by separating it with a colon (Listing 3.19).

Listing 3.19: Implementing the Hostname method (url.go)

```
// Hostname returns u.Host, stripping any port number if present.
func (u *URL) Hostname() string {     #B
    i := strings.Index(u.Host, ":")   #C
    if i < 0 {     #D
        return u.Host        #D
    }   #D
    return u.Host[:i]        #E
}
```

The method in Listing 3.19 searches in the `Host` field for a colon character and returns the hostname whether there is a colon in the `Host` field or not. If there's a colon, though, it returns the hostname part of the `Host` field. So, it's time for the `Port` method. Let's implement it in Listing 3.20.

Listing 3.20: Implementing the Port method (url.go)

```
// Port returns the port part of u.Host, without the leading colo
//
// If u.Host doesn't contain a port, Port returns an empty string
func (u *URL) Port() string {         #A
    i := strings.Index(u.Host, ":")   #B
    if i < 0 {     #C
        return ""  #C
    }   #C
    return u.Host[i+1:]      #D
}
```

The method in Listing 3.20 is similar to the `Hostname` method. It gets the port from the `Host` field whether there is a port in the `Host` field or not. It will return an empty string if there isn't a colon in the `Host` field.

Now that you're ready to run the tests:

```
$ go test
PASS
```

Phew! Well done.

3.4 Summary

You started the chapter with a request from the Wizards team and delivered a URL parser with idiomatic table-driven subtests. Well done! In the next chapter, you'll learn about refactoring and documentation to make code ready for release. Let's see what you have learned so far:

- Test helpers give the exact line number of an error
- Table-driven tests reduce complexity and repetition and help to cover edge cases
- Naming tests cases in a table-driven test are critical for finding the origin of failures
- Table-driven tests often use struct types but it's not a requirement
- Maps can shuffle the execution order of test cases
- Subtests give you superpowers for running tests in isolation
- Subtests allow writing concise failure messages
- Naming subtests is critical for the test summary and finding the origin of failures
- The run flag runs specific tests using regular expressions
- The failfast flag stops running other tests when a test fails
- The shuffle flag shuffles the execution order of top-level tests

4 Tidying Up

This chapter covers

- Writing testable examples
- Producing executable documentation
- Measuring test coverage and benchmarking
- Refactoring the URL parser
- Differences between external and internal tests

In this chapter, I'll teach you how to generate automatic and runnable documentation from code. I will show you how you can provide testable examples that never go out of date.

You'll learn about how to generate test coverage for the url package. I'll show you how to benchmark your code and give you a few tips about improving its performance.

Lastly, you'll learn to refactor the url package to make it more maintainable with the new knowledge you'll be acquiring in this chapter.

This is the last chapter of Part 1, and it's time to tidy up some left-overs from the previous chapters. Ready? Let's get started!

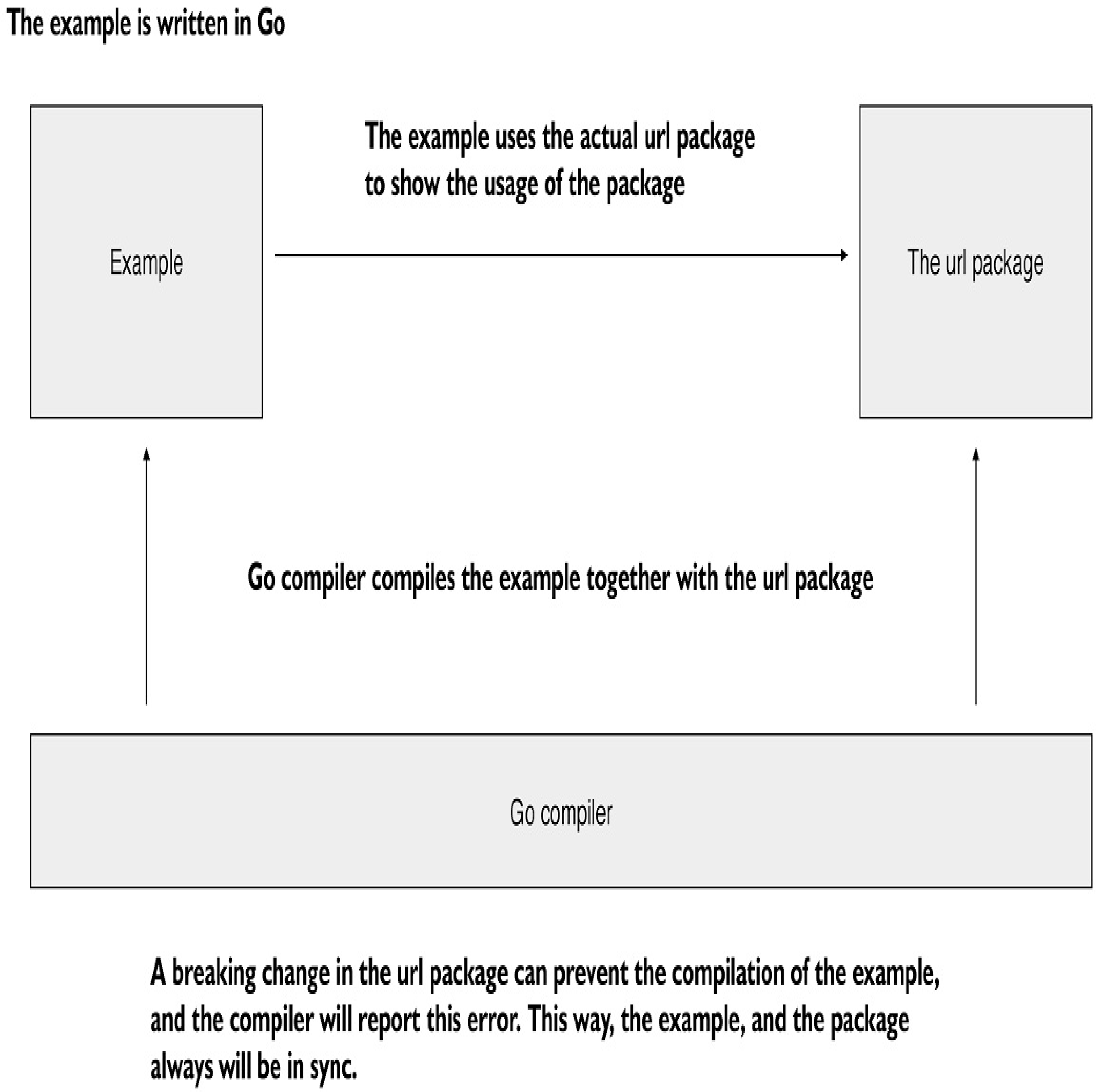
4.1 Testable Examples

These people again! The Wizards team, from the previous chapters. Oh my goodness. They're growing, and new people joined the team, and the existing team members got sick of explaining how the url package works to the new members. They ask you to provide sample code on using the package so the novice wizards can look at that instead of asking The Mighty Wizards.

How can you provide sample code to the Wizards? You might have thought you could give the Wizards a sample of code on how to use the package. But if you change the package's code in the future, then the sample code would become outdated. And, you would have to deliver the updated sample code every time you change the package's code.

You better start to look for a way that helps you to keep the samples updated whenever you change the package's code. This section will show you how to both document and verify code using *testable examples*. A testable example is like live documentation for code that never goes out of date (Figure 4.1). In other words, if the code changes in the future, the documentation will always be up to date.

Figure 4.1 Go compiler compiles an example together with the url package.



4.1.1 Writing a testable example

Remember, you want to demonstrate the usage of the url package from the point of view of the package's users.

Remember

An external package is a third-party package that you import. People can import the url package and use its exported functions and methods. To them, the url package will be an external package.

For example, the Wizards team will need to import the url package in their program before using it. So you can do the same thing:

- You can write sample code as if you were one of the developers in the Wizards team who wanted to use the package.
- The `url` package will appear to your sample code as an *external package*.

I'll deep dive what is an external package later in this chapter.

When might you want to write a testable example?

You often want to write a testable example showing other developers (and possibly yourself in the future!) how they can use a package's API. By API, I mean the exported identifiers from a package such as exported functions, methods, etc. The magic of testable examples is that they never go out of date.

Creating an example file

Although it's not a requirement, you can create an example file that starts with the `example_` prefix by convention. Let's first make a new empty test file called "`example_test.go`" in the same directory. You can do this as follows if you want to do it from the command line:

```
touch example_test.go
```

Note

Windows does not have a touch command. Instead, you can create an empty file in your favorite editor.

Since you will be *externally* testing the url package, you need to define a new package called `url_test` in the new file:

```
package url_test // defines a new package
```

The _test suffix in the package name has a special meaning in Go. You can use it when you want to write an *external test*. Since the `url_test` package is another package, you can only see the exported identifiers of the `url` package.

So you need to import the `url` package to test it:

```
import "github.com/inancgumus/effective-go/ch04/url"
```

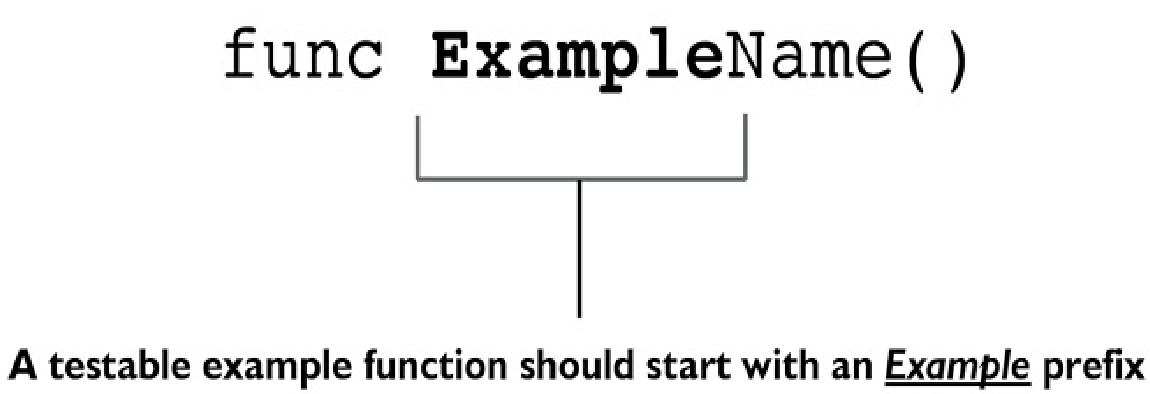
Great! You're almost ready to write a testable example.

What does a testable example look like?

Before going on further, let's take a look at an example function in Figure 4.2 to understand what a testable example function looks like. Unlike a test function, a testable example function doesn't have any input parameters, and it doesn't begin with a `Test` prefix. Instead, it starts with an `Example` prefix.

The testing package automatically runs testable examples and checks their results but doesn't let them communicate with it to report success or failure. That's why they don't take any input parameters like `*testing.T`.

Figure 4.2 The syntax of a testable example function. It starts with an Example prefix and does not take or return any parameters.



In summary, here is how you write a testable example:

1. You write a function that starts with the *Example* prefix.
2. You show an example usage of your package in the example function.
3. That's it!

Now you're almost ready to write your first testable example function.

Let's write a testable example function

Now you need to decide what code you want to demonstrate. I think the most important sample code you can provide to the Wizards is the usage of parsing a URL and getting its parts like scheme, host, etc.

So you can name the example function as `ExampleURL` because it will be demonstrating the usage of the URL type. In it, you will show parsing a raw URL string and printing the parsed URL (see the code in Listing 4.1).

Listing 4.1: Writing an example test (example_test.go)

```
func ExampleURL() {          #A
    u, err := url.Parse("http://foo.com/go")    #B
    if err != nil {          #C
        log.Fatal(err)       #C
    }   #C
    u.Scheme = "https"       #D
    fmt.Println(u) #E
    // https://foo.com/go
}
```

In Listing 4.1, first, you parse a URL with the `http` scheme. Then, you show that developers could change the scheme of a parsed URL if they want. As you can see, a testable example is like another code where all the features of Go are available.

So you can write code as you normally do:

- Parse a URL and get a parsed URL value.
- Check if there is an error.
- Show how to change the scheme of a URL as an example.

- Print and show the final state of the parsed URL value.

Congrats! You have your first testable example function. Let's try running it:

```
$ go test -run ExampleURL
testing: warning: no tests to run
```

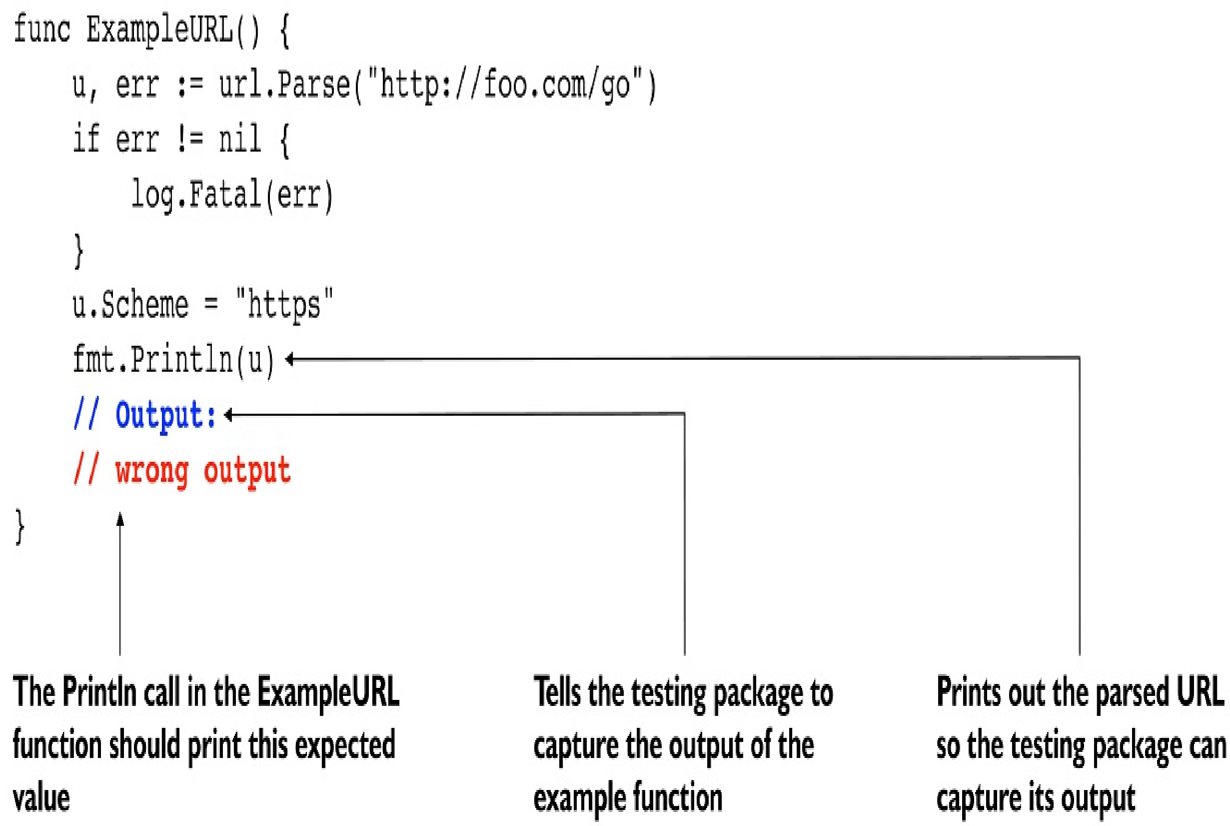
What?! No tests to run? Why can't you run it? This is because the sample code is for demonstration purposes only on how to use the URL package. Am I kidding? Why write a function if you can't run it? These questions are fair, and it's time to explain the reason behind writing an example function that you can't run.

Even if you can't run the example function in Listing 4.1, it's still valuable as the Go testing tool will compile it and check if it works. It doesn't have to be success or failure. If it compiles, it's good to go. That way, the example is never out of date. When you change the code in the url package, which may break the code in this example function, it will not compile. So you can go and update the sample code and keep it updated.

Let's add an expected output to the example

What if I told you that you could make the example function you wrote in Listing 4.1 more valuable? It would be awesome to demonstrate the usage of the url package and verify its output, wouldn't it? Good news, it's possible to do this! You can add a comment to the end of the function and tell the testing package what you expect to see from the example function (Figure 4.3). For this example, let's first include a wrong output and see what the testing package tells you when it fails.

Figure 4.3 The example function is now expecting an output, and the testing package will compare and verify the output of the sample code with this expected value.



1. Write an example function.
2. Add an `Output:` as a comment to the end of the function.
3. Then you provide the expected output also in a comment.

That's it! When you do this, the testing package can automatically verify if the example produces the *expected output* and print a failure message if the output doesn't match the expected output. Now, it's time to run the example function. You can run it just like other tests using the `go test` command.

Here's the output after you run it:

```
$ go test -run ExampleURL
--- FAIL: ExampleURL
got:
&{https foo.com go}
want:
wrong output
```

The example function was expecting an incorrect output. That's why you see a test failure here. The test will pass if you change the output comment as follows:

```
// Output:
// &{https foo.com go}
```

When you have a single line of output, you can also write it as follows:

```
// Output: &{https foo.com go}
```

For multiple lines of output, you can add as many comments as you like. For example:

```
// Output:
// Somewhere over the rainbow
// Way up high
// And the dreams that you dream of
// Once in a lullaby, oh
```

Now the novice Wizards team members can easily look at the code in Figure 4.3 and see how they can use the url package. Whenever you change the package's code, the testable example will verify the changes like a test function. The example function is now both a test function and also acts as documentation.

Unordered output

Let's say you want to write an example function for a function that returns random numbers. So every time the testing package ran the example, its output would change and fail miserably.

Fortunately, for these cases, the testing package allows you to add another comment instead of the "`// Output:`" comment as you saw earlier. Unsurprisingly, it looks like: "`// Unordered output:`".

Let me show you an example. The `Perm` function of the `rand` package returns a slice of random numbers. If you are writing an example for the `Perm` function, you can use the unordered output comment as follows:

```
func ExamplePerm() {
```

```
    // seeds the random number generator with
    // the current time to get random numbers
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    // Perm will return a slice of random numbers with three elem
    for _, v := range r.Perm(3) {
        fmt.Println(v)
    }
    // Unordered output:
    // 2
    // 0
    // 1
}
```

4.1.2 Self-Printing URLs

The Wizards team is using the url package in their program. While redirecting the requests to a new location, they want to log the parsed URLs as well. Other gophers can also find this new feature helpful for debugging purposes, print it to the console, or store it in a data store. But, the Wizards couldn't find an easy way to print a parsed URL using the url package.

For example, let's say you want to print the following URL value:

```
u, _ := Parse("https://foo.com/go")
fmt.Println(u)
```

If you were to run the code, you would see the following output:

```
&{https foo.com go}
```

It's not helpful, is it?

You can also see this problem in the output of Listing 4.2. The example function showed that the correct output would be "`&{https foo.com go}`". But it doesn't look good and is not easily comprehensible. The novice developers couldn't understand the examples and are still asking the Wizards. So they need you to add the ability to print a parsed URL. Let's talk about how you can make the output better for human beings.

Adding a String method

Wouldn't it be great if you had a method that could print a `URL` in a human-readable format? What about writing a method called `String` on the `URL` type? This method can read a parsed URL's fields and return an easy-to-read string.

For now, let's return a dummy string value:

```
func (u *URL) String() string {
    return "fake"
}
```

No worries. You will implement the `String` method soon.

If you remember from the previous chapters, you used the `Parse` function to parse a URL. Instead of doing this, why not create a `URL` value from scratch and test it? This way, you can write a simpler test and no longer need to check the error value from the `Parse` function.

Tip

In unit tests, it's better to test code in isolation as much as you can.

Listing 4.2: Testing the String method (url_test.go)

```
func TestURLString(t *testing.T) {    #A
    u := &URL{     #B
        Scheme: "https",     #B
        Host:   "foo.com",   #B
        Path:   "go",        #B
    }   #B
    got, want := u.String(), "https://foo.com/go"         #C
    if got != want {         #C
        t.Errorf("%#v.String()\ngot  %q\nwant %q", u, got, want)
    }   #C
}
```

The test in Listing 4.2 expects the `URL` value to produce `https://foo.com/go` when you call its `String` method:

- Creates a parsed URL from scratch.
- Calls its `String` method.

- And, compares the returned value to the raw URL: `"https://foo.com/go"`.

If you noticed, the test uses a different kind of failure message format, this time:

```
"%#v.String()\ngot  %q\nwant %q"
```

The failure message separates the actual and expected values by *newline characters (*\n*)* to produce a more readable failure message:

```
$ go test -run TestURLString
--- FAIL: TestURLString
     &url.URL{Scheme:"https", Host:"foo.com", Path:"go"}.String()
     got "fake"
     want "https://foo.com/go"
```

I think the failure message is pretty readable and tells you what you expected and what you got. Now it's time to implement the `String` method on the URL type for real. The `String` method in Listing 4.3 retrieves and combines a `URL`'s fields to produce its string representation.

Listing 4.3: Implementing the String method (url.go)

```
import "fmt"       #A

// String reassembles the URL into a URL string.
func (u *URL) String() string {       #B
    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

The `Sprintf` function is similar to the `Errorf` and `Fatalf` method you used earlier:

- It takes a *formatting specifier* and a variable number of *values*.
- Then it replaces the placeholders (more formally known as *verbs*) in the formatting specifier with these values, one by one.
- So it will replace the first placeholder `%s` with the Scheme field.
- The second placeholder with the `Host` field.
- And the last placeholder with the `Path` field.

The test will pass when you run it:

```
$ go test -run TestURLString
PASS
```

Fixing the example

Well done! But not so fast! There is a little problem with your tests. The example function that you wrote earlier is now failing:

```
$ go test
--- FAIL: ExampleURL
got:
https://foo.com/go
want:
&{https foo.com go}
```

Previously, since you ran the test with the `run` flag, the testing package only ran the `TestURLString` test, not the other tests. So that's why you didn't see the example function was failing. You can easily fix the example if you include the string representation of the URL under the `Output` comment as in Listing 4.4.

Listing 4.4: Fixing the example (example_test.go)

```
func ExampleURL() {
    u, err := url.Parse("http://foo.com/go")
    ...
    u.Scheme = "https"
    fmt.Println(u) #A
    // Output:
    // https://foo.com/go    #B
}
```

All tests will pass when you run them. You can now print a `URL` value like this:

```
u, _ := Parse("https://foo.com/go")
fmt.Println(u)
// https://foo.com/go
```

Great! You have successfully reconstructed the URL string from a parsed

URL value. Now it looks pretty readable. By the way, have you noticed the magic? Something is going on here. How could the `Println` function automatically run the `String` method? It did so because the `Println` function detected that the URL type was a `Stringer,` and it automatically called its `String` method. No worries. I'll talk about this phenomenon next.

The magic of interfaces

It's time for explaining the magic of the `String` method, or in other words, the `Stringer` interface. It's one of the most common and basic interfaces in the Go Standard Library and allows you to customize the string representation of a type. When you add a `String` method to the URL type, you make it a `Stringer`.

In Go, when a type satisfies an interface, we often say: "*Type X is a Y*." In this case, we would say: "*URL is a Stringer*." In other words, the URL type satisfies the `Stringer` interface.

Here is what the `Stringer` interface looks like:

```
type Stringer interface {
    String() string
}
```

So the functions or methods that recognize the `Stringer` interface can use the `String` method when printing a URL value. For example, the printing methods such as `Printf` or `Println` will call the `String` method of a `URL` value and print a prettier representation of it.

So you don't need to call the `String` method yourself:

```
u, _ := Parse("https://foo.com/go")
fmt.Println(u.String())
fmt.Println(u)
```

```
Both calls to the Println will print the same thing:
```

```
https://foo.com/go
https://foo.com/go
```

Cool!

4.1.3 Runnable examples

Before starting, heed on! As of Go version 1.13, the godoc tool doesn't come by default with Go. You can install the latest version as follows if you don't have it:

```
go install golang.org/x/tools/cmd/godoc@latest
```

As you saw in the previous section, testable examples (or example functions) demonstrate the usage of code and verify it. But the benefits of writing a testable example don't end there. There is another benefit.

When you write a testable example, it also serves as executable documentation for code. So the novice team members of the Wizards team don't have to look at the testable example's source code. And instead, they can view the url package's documentation and run the package's example code on their browser as well.

For example, let's take a look at the URL type's documentation in Figure 4.4.

Figure 4.4 An executable documentation for the URL type

- The comment you wrote in the code before the definition of the URL type appears as a description at the top of the documentation, next to the "*type URL*" heading.
- The code definition of the URL struct type appears just below the type's

description.
- A runnable example appears right below the URL type definition.

So developers can see what the URL type looks like and play with it using the Run button at the bottom right. They can also format the code and share it with others.

Running the examples locally

The documentation you saw in Figure 4.4 is from the Go Doc server. The Go team serves it online but you can run the documentation server on your machine as well. Let's take a look at how you can do that.

First, you need to run the following command:

```
$ godoc -play -http ":6060"
```

This command will run a local server that listens on port 6060. Then you can go to the following URL in your browser and see it in action:

```
http://localhost:6060/pkg/github.com/inancgumus/effective-go/ch04
```

If you run the command without the `-play` flag, you won't be able to run the code in the testable example. But you can still see the code and its output.

Adding more examples

The novice wizards can now see the example function and play with it. They no longer need to ask the mighty wizards how they can use the url package. But if you would like to provide more examples, you can also do that. It will automatically appear in the URL type documentation.

For example, you can document the URL type's fields after parsing a raw URL. To do that, let's declare one more example function in listing 4.5.

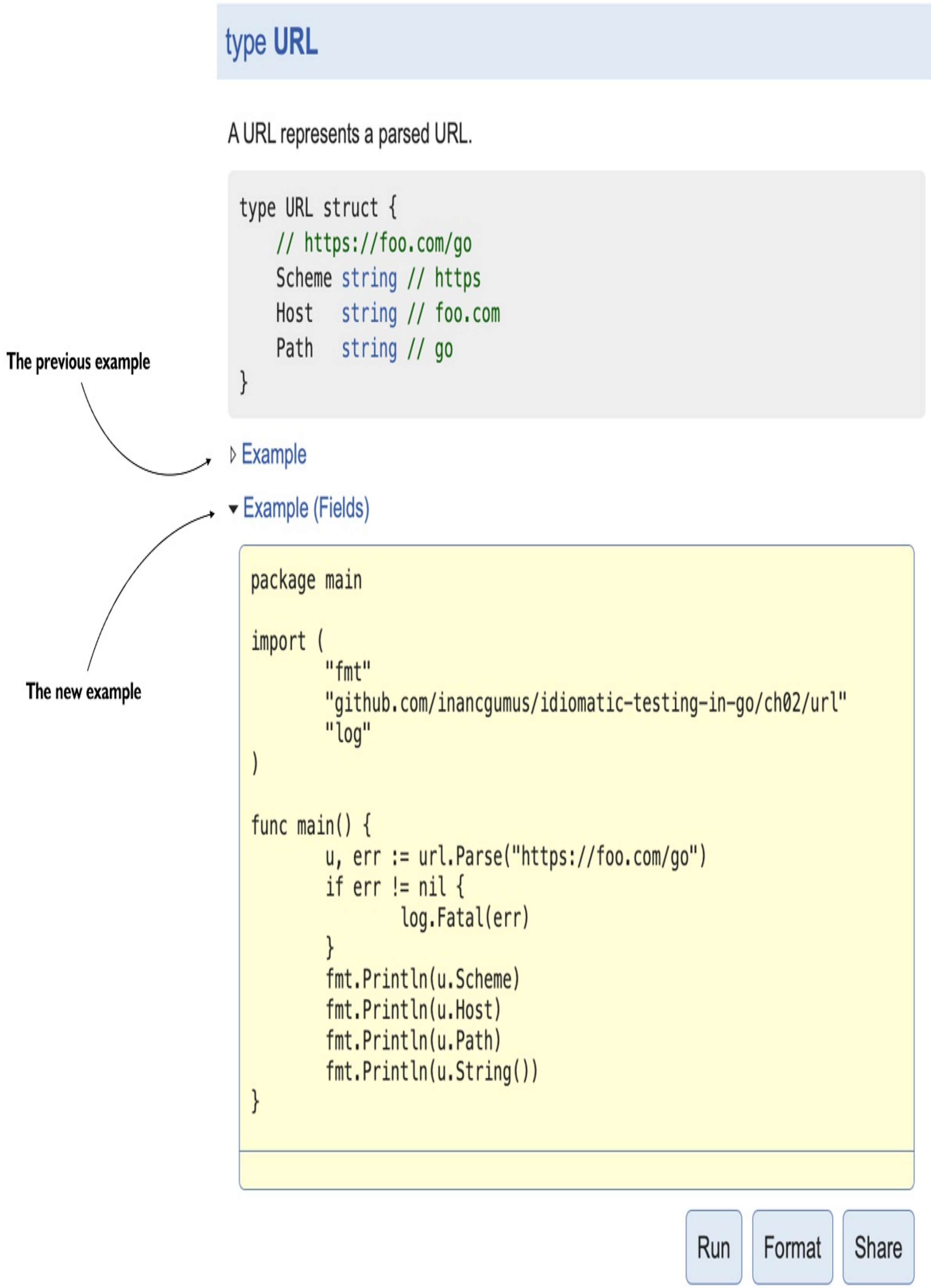
Listing 4.5: Adding additional example test (url_test.go)

```
func ExampleURL_fields() {   #A
```

```
    u, err := url.Parse("https://foo.com/go")
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(u.Scheme)
    fmt.Println(u.Host)
    fmt.Println(u.Path)
    fmt.Println(u)
    // Output:
    // https
    // foo.com
    // go
    // https://foo.com/go
}
```

If you're still running the `godoc` server, you can refresh the documentation on your browser to automatically update the documentation with the new example. The new example will appear next to the previous example as in Figure 4.5.

Figure 4.5 The executable documentation of Example (Fields)



type URL
A URL represents a parsed URL.
type URL struct {
    // https://foo.com/go
    Scheme string // https
    Host   string // foo.com
    Path   string // go
}
The previous example
Example
The new example
Example (Fields)
package main

import (
	"fmt"
	"github.com/inancgumus/idiomatic-testing-in-go/ch02/url"
	"log"
)

func main() {
	u, err := url.Parse("https://foo.com/go")
	if err != nil {
		log.Fatal(err)
	}
	fmt.Println(u.Scheme)
	fmt.Println(u.Host)
	fmt.Println(u.Path)
	fmt.Println(u.String())
}
Run
Format
Share

- When you want to show additional examples of the same type, add a lowercase suffix with an underscore character.
- If you noticed, the example function in listing 2.39 has a lowercase suffix: `_fields`.
- This suffix allows you to show the new example next to the previous one.

Naming conventions for testable examples

There are some specific naming conventions for testable examples. You can find all of them below. You can also find more examples in the official Go blog: https://blog.golang.org/examples.

Signature	Explanation
`func Example()`	Provides an example for the whole package. In this case, there should only be a single testable example in an example test file.
`func ExampleParse()`	Provides an example for the `Parse` function.
`func ExampleURL()`	Provides an example for the `URL` type.
`func ExampleURL_Hostname()`	Provides an example for the `Hostname` method of the `URL` type.

4.1.4 Wrap up

Congrats! You learned a lot of new knowledge. The Wizards team had asked you to provide documentation for using the url package to their novice team members. You went further and provided automatically updating and executable documentation where they can play on their browser. Well done!

Note

You can learn more about example functions in the link:

https://blog.golang.org/examples.

Let's summarize what you've learned so far:

- Testable example functions reside in an external test package and demonstrate the usage of a package from the point of view of the package's users. They start with an `Example` prefix and provide up-to-date documentation.
- An `"Output:"` comment makes a testable example function to act as a test and verify the example's output.
- `godoc` server allows you to see and run example functions on a browser.

4.2 Test coverage

Imagine a radar system that can detect objects within a range of 500 miles. This range is the coverage capability of the radar system. Similarly, tests cover code within a range and use lines of code instead of miles. This section will show you how to measure *test coverage*. It's a way of cross-checking to see which parts of code that tests are verifying, or in other words, covering.

The coverage problem

Some of your team members noticed that you had a test for the `Parse` function but didn't verify all kinds of URLs, so they are worried about possible bugs. They noticed that you were only testing for a single URL:

```
func TestParse(t *testing.T) {
    const rawurl = https://foo.com/go
    u, err := Parse(rawurl)
    ...
}
```

The URL in the test function has a scheme, host, and path: `"https://foo.com/go"`. But what about URLs without these parts such as "`https://`" or "`foo.com`"? You're not testing these URLs. So it looks the test may not cover every aspect of the `Parse` function yet. Could you find this problem before they did? Yep, sure, you could.

4.2.1 Measuring test coverage

It's time to get practical and measure test coverage. But before that, you need to let the test tool analyze the code in the url package and generate a coverage profile for it as follows:

```
$ go test -coverprofile cover.out
PASS
coverage: 94.1% of statements
```

The test tool analyzed the code of the url package, generated a *coverage profile*, and saved it in a text file named "`cover.out`". The tool also tells you that all tests passed, and tests cover 94.1% of the code. But where is the missing piece? Why do they not cover the remaining 5.9%?

You can look at the generated file and see the lines that end with zero to find that out. You will find that the tests do not verify the code on lines between 20 and 22:

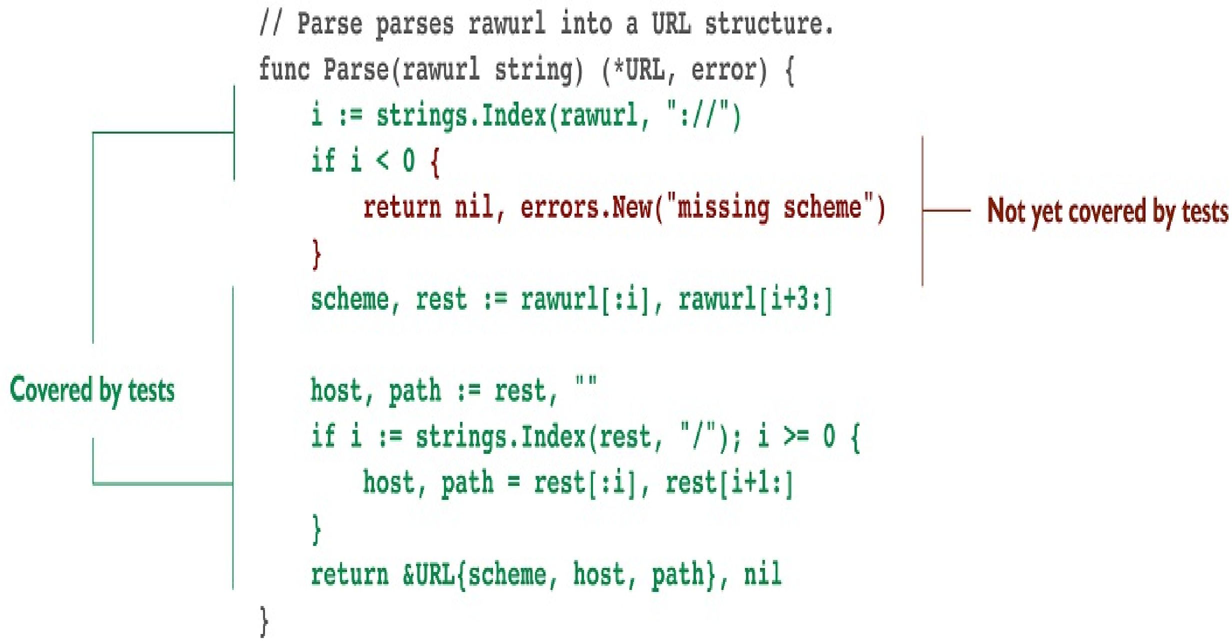
```
...url.go:20..22...0
```

Don't worry. You don't need to check the coverage profile to determine which parts of the code are not yet covered. Instead, you can use a tool called the *coverage tool*. You can feed it with the coverage profile file to see the uncovered areas of your code like this:

```
$ go tool cover -html=cover.out
```

This command should automatically open a window in your default browser and show you the coverage report similar to Figure 4.6.

Figure 4.6 The report shows that tests do not cover the missing scheme error.



You better open the coverage report on your machine because this book will be printed in gray color. Let's figure out what's going on:

- The green lines are the areas in the code where tests cover.
- The red lines are where tests don't cover.
- The gray lines are untracked by the coverage tool.

It's as simple as that! It seems like you have pretty good test coverage. You're testing almost every aspect of the code except one line.

So why does the `TestParse` test not cover the missing scheme error? This is because the test only verifies the *happy path*. So it doesn't verify edge cases like a missing scheme of a URL. This is a good time to create a table-driven test in Listing 4.6 so you can test for all the edge cases.

Listing 4.6: Adding a test for covering edge-cases (url_test.go)

```
func TestParseInvalidURLs(t *testing.T) {
    tests := map[string]string{       #A
        "missing scheme": "foo.com",  #B
        // test cases for other invalid urls
    }
    for name, in := range tests {
```

```
        t.Run(name, func(t *testing.T) {
            if _, err := Parse(in); err == nil {
                t.Errorf("Parse(%q)=nil; want an error", in)
            }
        })
    }
}
```

- For now, the test in Listing 4.6 has only one test case for verifying the missing scheme error. Hang on! You'll add more test cases for all the other errors soon.
- As you can see, a table-test doesn't have to include a struct type all the time. You can use a plain-old string as well.
- Here, the test cases are in a map, and each one has a name for an error condition and a URL to pass to the `Parse` function and test it with.

Let's take a look at the coverage profile:

```
$ go test –cover
PASS
coverage: 100.0% of statements
```

Great! You have now covered every line of code with tests. Your team members will be proud of you. By the way, did you notice that you used a different flag? You don't have to generate a coverage profile with the `cover` flag. The downside is that it doesn't show you the actual code, but now you have 100% coverage; it's no longer a problem.

Avoiding the browser

If you don't want the cover tool to open a browser window, you can save the coverage report to an HTML file like this:

```
$ go tool cover -html=cover.out -o coverage.html
```

You can also see the coverage of each function from command line like this:

```
$ go tool cover -func=cover.out
url.go:17:  Parse       100.0%
url.go:32:  Hostname    100.0%
url.go:43:  Port        100.0%
```

```
url.go:52:  String       100.0%
total:    (statements)  100.0%
```

4.2.2 Test coverage != Bug-free

You achieved 100% test coverage, and you feel super cool. Your code is bug-free, yep? I have bad news; it's not. So sorry for your loss. The truth is test coverage can only show which parts of code that tests cover, but it cannot find the bugs for you. Let's discuss this problem.

Empty scheme

For example, let's add one more test case to the `TestParseInvalidURLs` test (*which you wrote earlier in Listing 4.6*) and see why gravity always pulls you down:

```
"empty scheme": "://foo.com",
```

The empty scheme test case will verify a URL without a scheme. The weird thing is that it has a scheme signature (`://`), so now let's see what the test will tell you:

```
$ go test -run TestParseInvalidURLs
--- FAIL: TestParseInvalidURLs/empty_scheme
    Parse("://foo.com")=nil; want an error
```

What? You might say: "*My code has 100% test coverage and is bug-free! Yippeee!*". Told you! Test coverage cannot guarantee that your code is bug-free. The test output tells you the `Parse` function cannot handle such a strange URL. Fortunately, the fix is easy (Listing 4.7).

Listing 4.7: Fixes the scheme bug (url.go)

```
func Parse(rawurl string) (*URL, error) {
    i := strings.Index(rawurl, "://")
    if i < 1 {     #A
        return nil, errors.New("missing scheme")
    }
    ...
}
```

Previously, the `Parse` function searched for a scheme signature in the `rawurl` and checked its existence (`i < 0`). It now checks if the signature starts from the second character (`i < 1`). So a URL's scheme should have a character before the scheme signature. The test will pass when you run it.

The String method

There is another problem with the `String` method and test you wrote earlier. You can see them in the previous listings 4.3 and 4.4 (*Testable Examples section's Self-Printing URLs heading*).

You can see that the `String` method doesn't care whether a URL doesn't have a scheme, host, or path, and it just prints a URL, and its test doesn't check for URLs without these parts:

```
func (u *URL) String() string {
    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

Let's rewrite the `TestURLString` test from scratch by converting it to a table-test and adding edge-cases in Listing 4.8.

The test:

- Creates a test table that has input `*URL` values and expected string values for those URLs.
- Runs the `String` method on each `*URL` value in the table and checks if it gets the expected string value.

Listing 4.8: Adding edge-case tests to TestURLString (url_test.go)

```
func TestURLString(t *testing.T) {
    tests := map[string]struct {
        url  *URL           #A
        want string         #B
    }{
        "nil url":   {url: nil, want: ""},
        "empty url": {url: &URL{}, want: ""},
        "scheme":    {url: &URL{Scheme: "https"}, want: "https://
        "host": {
```

```
            url:  &URL{Scheme: "https", Host: "foo.com"},
            want: "https://foo.com",
        },
        "path": {
            url:  &URL{Scheme: "https", Host: "foo.com", Path: "g
            want: "https://foo.com/go",
        },
    }
    for name, tt := range tests {
        t.Run(name, func(t *testing.T) {
            if g, w := tt.url, tt.want; g.String() != w { #C
                t.Errorf("url: %#v\ngot:  %q\nwant: %q", g, g, w)
            }
        })
    }
}
```

- Here, a couple of test cases verify the `String` method with `nil` and empty `*URL` values etc.
- The first test case is "`nil url`" which verifies what happens when you call the `String` method on a `nil *URL` value.

Let's begin with it:

```
$ go test -run 'TestURLString/nil url'
--- FAIL: TestURLString/nil_url
panic: runtime error
```

Oops! The test panicked. This is because the `String` method tried to read the fields of a `nil *URL` value. You can't read fields, but you can call methods on a `nil` value! It's weird but handy, especially in this case: Even a `nil` url can print itself!

To do that, you can return an empty string if you detect that a `*URL` value is `nil` as follows:

```
func (u *URL) String() string {
    if u == nil {
        return ""
    }
    return fmt.Sprintf("%s://%s/%s", u.Scheme, u.Host, u.Path)
}
```

This will fix the panic, and the test will pass. Let's continue with the remaining test cases:

```
$ go test -run TestURLString
--- FAIL: TestURLString/empty_url
    url: &url.URL{Scheme:"", Host:"", Path:""}
        got:  ":///"
        want: ""
--- FAIL: TestURLString/scheme
    url: &url.URL{Scheme:"https", Host:"", Path:""}
        got:  https:///
        want: https://
--- FAIL: TestURLString/host
    url: &url.URL{Scheme:"https", Host:"foo.com", Path:""}
        got:  https://foo.com/
        want: "https://foo.com"
```

You now have brand new failing test cases, and they all fail because of the same reason: The `String` method should not include empty `*URL` fields. You can easily fix this problem by checking each field and returning the non-empty fields (Listing 4.9).

Listing 4.9: Fixing the String method (url.go)

```
func (u *URL) String() string {
    if u == nil {
        return ""
    }
    var s string   #A
    if sc := u.Scheme; sc != "" {     #B
        s += sc
        s += "://"
    }
    if h := u.Host; h != "" {         #B
        s += h
    }
    if p := u.Path; p != "" {         #B
        s += "/"
        s += p
    }
    return s
}
```

- Creates a string variable to return in the end.

- Adds each field to the variable if the field is not empty.
- Returns the variable only with non-empty fields.

You finally have fixed the String method, and the test will pass when you run it. Congrats!

How does Go run a method on a nil value?

A method is a function that takes the receiver as a hidden first parameter.

Behind the scenes, the `String` method look as follows:

```
String(u *URL) string { /* code */ }
```

Let's say you have a `nil *URL` value as follows:

```
var u *URL
u.String()
```

`u.String()` is equal to `(*url.URL).String(u)`. And, `(*url.URL)` tells the compiler the receiver's type. It's the `*URL` type in the `url` package. The compiler goes there and finds the method.

When you have a `nil *URL` value, behind the scenes, the compiler passes it as follows:

```
(*url.URL).String(u)  // String receives a nil *URL
```

In the end, the `String` method receives the `nil *URL` value.

4.2.3 Wrap up

The test coverage should not be your goal but is a helpful guidance. Above 80% coverage is usually more than enough, but, as always, it depends.

Let's summarize what you've learned so far:

- `go test -coverprofile cover.out` generates a coverage profile and saves it to a file named `cover.out`.

- `go tool cover -html=cover.out` generates an HTML document by reading the coverage profile in `cover.out` and opens a new browser window and shows the coverage report.
- `go test -cover` shows the test coverage in percentage without needing a coverage profile file.
- Test coverage helps you find untested code areas, but it doesn't guarantee 100% bug-free code.

Note

You can learn more about test coverage in the link: https://blog.golang.org/cover.

4.3 Benchmarks

The url package you wrote is a part of the Go Standard Library, and you might expect that millions of Go developers will use it all the time. For example, the Wizards team's program may receive billions of web requests and create `*URL` values for each request. So you might want to optimize the url package and make the Wizards happier.

Benchmarking is the process of running the same code repeatedly and measuring how it performs on average because it has to be statistically significant. Roughly speaking, statistical significance is about seeing whether an outcome happens by chance or not. Imagine flipping a coin hundreds of times. You will likely get ~50% heads and ~50% tails. But if you had flipped it less, for example, ten times, you wouldn't probably get these ratios. Similarly, it would be best if you benchmarked code enough times to make it statistically significant.

Warning

The art and science of optimization and properly benchmarking code is beyond the scope of this book. So I can only show you how to write benchmarks in Go.

4.3.1 Optimizing the String method

You can use benchmarks to optimize the url package. The difference between tests and benchmarks is that benchmarks measure code performance and tests verify code correctness. You measured the test coverage of the `String` method in the previous section. What about measuring its performance now? Sounds good to me.

So let's measure the average performance of the `String` method by writing a benchmark function. A benchmark function is similar to a test function but starts with a `Bench` prefix instead of a `Test` prefix and takes a `*testing.B` parameter instead of `*testing.T` (Listing 4.10).

Listing 4.10: Writing a benchmark for the String method (url_test.go)

```
func BenchmarkURLString(b *testing.B) {         #A
    u := &URL{Scheme: "https", Host: "foo.com", Path: "go"}
    u.String()     #C
}
```

As you can see, you can put a benchmark function in the same test file as your other tests. You create a `*URL` value in the function, and then you call the `String` method to measure its performance. Easy-peasy.

While in the url package's directory, you can use the `bench` flag and pass it a *dot* to match every benchmark and measure the performance of the package (*which is the url package*) in general like this:

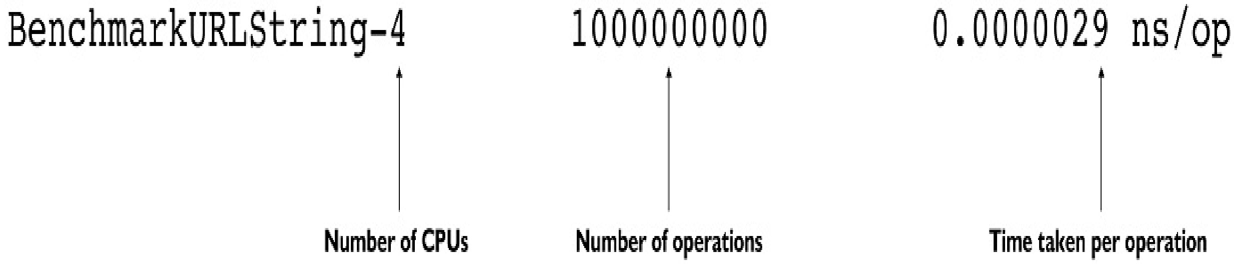
```
$ go test -bench .
BenchmarkURLString-4    1000000000    0.0000029 ns/op
```

Tip

Just like the `run` flag, the `bench` flag also accepts regular expressions. You can even separate multiple regular expressions with slash characters to match sub-benchmarks. On the other hand, a *dot* has a special meaning for the `bench` flag and matches every benchmark in the current package.

Let's see what these gibberish numbers mean in Figure 4.7.

Figure 4.7 The meanings of the fields in the benchmark report.



- `BenchmarkURLString-4` means that the testing package ran the benchmark code using 4 CPU cores because there are 4-cores on my old Macbook (which I'm using for writing this book).
- `1000000000` means the benchmark function ran one billion times.
- `0.0000029 ns/op` means each function call (`op`) took 0.0000029 nanoseconds.

Ideally, you should run benchmarks on a dedicated machine to isolate the results. Otherwise, external and internal factors can affect the results.

Running only the benchmarks

The testing package will still run your tests together with benchmarks even when you use the bench flag. You can see that is happening if you had used the verbose flag:

```
$ go test -v -bench .
=== RUN   TestParse
--- PASS: TestParse
...
BenchmarkURLString
...
```

This is often not a problem but you can use the following trick to avoid running the tests if your tests take a lot of time and you don't want to wait:

```
$ go test -run=^$ -bench .
```

This regular expression (^$) will tell the runner not to match any tests, so that won't run them, and only the benchmarks will run.

4.3.2 Proper benchmarking

I think something fishy was going on in the previous example because the testing package runs a benchmark function up to one billion times, and the default benchmark running time is one second. For a dummy but a practical way to see how many times the testing package calls your benchmark function, you can print a message in the benchmark function:

```
func BenchmarkURLString(b *testing.B) {
    b.Log("called")
    ...
}
```

As you can see, you can use the `Log` method because almost all the methods of the `*testing.T` type that you learned before are also available in the `*testing.B` type. This is because they share common functionality to some extent. Anyway, let's save the file and rerun the benchmark:

```
$ go test -bench .
    ...
    Called
    Called
    Called
    called
    called
    called
    ...
```

The benchmark ran six times instead of a billion times! Why is that so? This is because your benchmark function returned so fast that the testing package couldn't adjust itself. This is clearly not statistically significant to measure the performance of the `String` method.

Helping the runner adjust itself

I have good news as usual. A field called `N` in the `*testing.B` type helps the benchmark runner adjust itself. To do that, you can call the `String` method in a loop for `N` times to make the result statistically significant (Listing 4.11).

Listing 4.11: Fixing the benchmark (url_test.go)

```
func BenchmarkURLString(b *testing.B) {
    b.Logf("Loop %d times\n", b.N)              #A
```

```
    u := &URL{Scheme: "https", Host: "foo.com", Path: "go"}
    for i := 0; i < b.N; i++ { #B
        u.String()
    }   #B
}
```

- The benchmark creates a `*URL` value once because it only wants to measure the performance of the `String` method.
- The runner can now adjust the `b.N` variable each time it calls the benchmark function to measure its performance properly.
- You also log the `b.N` variable to see how the benchmark runner makes adjustments.

The runner will call the `String` method in the loop for `b.N` times and report its performance:

```
$ go test -bench .
BenchmarkURLString-4    3704509    313.9 ns/op
    ...
    Loop 1 times
    Loop 100 times
    Loop 10000 times
    Loop 1000000 times
    Loop 3704509 times
ok    github.com/inancgumus/effective-go/ch04/url    2.240s
```

The result is now meaningful:

- The runner called the code in the benchmark function about 4 million times.
- And each call took roughly 300 nanoseconds.

You might say: "*But there are five calls to the benchmark function, and the total number of iterations is about five million times, not four!*" You really have keen eyes, and you're right, but the runner reported the result only from the last loop because the other loops only adjust the runner.

Measuring memory allocations

Benchmarking is not only about measuring the operations per second. You can also measure the memory allocations of your code, and often these two aspects are correlated. To do that, you can call the `ReportAllocs` method like this:

```
func BenchmarkURLString(b *testing.B) {
    b.ReportAllocs()
    ...
}
$ go test -bench .
BenchmarkURLString-4    7178090    151.5 ns/op    56 B/op    3 al
```

The "B/op" column shows how many bytes were allocated in total per operation. So the code in the benchmark allocates 56 bytes.

And the "allocs/op" column shows how many memory allocations calls to your operating system happened. So the code in the benchmark made 3 allocation calls to the operating system. Memory allocation means getting more memory from the operating system and reducing the available memory to your and other programs on your system.

If you're curious, you can read the source code of the Go memory allocator at the link: https://go.dev/src/runtime/malloc.go. There is another good discussion at the link: https://go.dev/doc/diagnostics.

4.3.3 Comparing benchmarks

Sometimes, a single result may be enough, but often it is not. So you usually need to compare the older performance results with the new ones to see if any optimizations you make are worth the hassle.

Here's the plan:

1. Save the benchmark result of the `String` method.
2. Find out how you can optimize it.
3. Remeasure it and compare it with the previous result.

You already measured the performance of the `String` method, but you didn't save the benchmark result to a file. So you'll first do that. Only then you'll start optimizing the `String` method. In the end, you will compare the previous performance result with the new one after you made the optimization.

1. Saving the old benchmark result

Let's start with the first step. Since the environmental factors can bend the benchmark results, you can use a flag called `count` to run the same benchmark multiple times. Then you can save the benchmark result to a file like this:

```
$ go test -bench . -count 10 > old.txt
```

There is not a single magical value that you can provide to the `count` flag, so ten times is not mandatory.

2. Optimizing the String method

Let's move on to the second step. The current `String` method in Listing 4.9 uses string concatenation to combine the fields of a `*URL` value. It can be problematic because Go creates a new string value every time you combine string values. Combining string values works in most cases, but it can be pretty inefficient if the `*URL` values reside in a hot path. So you may be producing new string values and increasing the pressure on the Go Garbage Collector without realizing it.

Users that have come to Go from other languages such as Java know that string concatenation operators such as += are (maybe were) very expensive.

So instead of creating a lot of string values in the process, you can use a type called `Builder` from the `strings` package to do this efficiently. It's simply a buffer where you can add multiple string values and get a single string value at the end. You will add the field values to the buffer and return it as a single value at the end (Listing 4.12).

Listing 4.12: Optimizing the String method with the Builder (url.go)

```
func (u *URL) String() string {
    if u == nil {
        return ""
```

```
    }
    var s strings.Builder    #A
    if sc := u.Scheme; sc != "" {       #B
        s.WriteString(sc)    #B
        s.WriteString("://") #B
    }   #B
    if h := u.Host; h != "" {           #B
        s.WriteString(h)     #B
    }   #B
    if p := u.Path; p != "" {           #B
        s.WriteByte('/')     #B
        s.WriteString(p)     #B
    }   #C
    return s.String()        #C
}
```

As you can see in Listing 4.12:

- The method makes a new buffer.
- Then it adds the URL fields to the buffer.
- Finally, it returns the buffer as a string value by calling its `String` method.

Tip

You can check out the link if you want to learn more about the `Builder` type: https://stackoverflow.com/questions/1760757/how-to-efficiently-concatenate-strings-in-go/47798475#47798475

3. Comparing the benchmark results

It's time to measure the performance of the optimized `String` method. To do that, you can run the same benchmarks and save the result to a new file like this:

```
$ go test -bench . -count 10 > new.txt
```

Alright! It's now time to compare the benchmark results, but it would be hard to do that manually. No worries. There is an external tool called `benchstat` to compare benchmark results. Let's install it on your machine as follows:

```
$ go install golang.org/x/perf/cmd/benchstat@latest
```

You can now compare the old and new benchmark results using the `benchstat` tool like this:

```
$ benchstat old.txt new.txt
name          old time/op  new time/op  delta
URLString-4   273ns ± 8%   150ns ± 2%   -44.82%  (p=0.000 n=10+9)
```

You have improved the performance of the `String` method by ~45%. The previous version of the String method ran in 273ns and the new one in 150ns. Unfortunately, the optimization you made to the `String` method can be inaccurate. As I explained before, this book is not about performance optimization. At least you know how to create and compare benchmarks! Well done!

Sub-benchmarks

Measuring the performance of the String method with different URL values can give you more accurate results. You can use sub-benchmarks to do that. Similar to subtests, you can run multiple sub-benchmarks under a single benchmark function.

For example:

```
func BenchmarkURLString(b *testing.B) {
    var benchmarks = []*URL{
        {Scheme: "https"},
        {Scheme: "https", Host: "foo.com"},
        {Scheme: "https", Host: "foo.com", Path: "go"},
    }
    for _, u := range benchmarks {
        b.Run(u.String(), func(b *testing.B) {
            for i := 0; i < b.N; i++ {
                u.String()
            }
        })
    }
}
```

The `BenchmarkURLString` has three sub-benchmarks, and each measures the performance of the `String` method using different URL values. The benchmark function runs these sub-benchmarks using the `Run` method of the `*testing.B` type. It's very similar to the `Run` method of the `*testing.T` type. So everything you learned about subtests before also applies to sub-benchmarks.

4.3.4 Wrap up

Let's summarize what you've learned so far:

- Performance optimization can be tricky and is a science in itself.
- Benchmark functions start with a `Benchmark` prefix and take a `*testing.B` parameter.
- Almost all the methods of the `*testing.T` type are also available for the `*testing.B` type.
- The `bench` flag is similar to the run flag and takes a regular expression to match benchmarks and sub-benchmarks.
- The `count` flag can run the same benchmark functions many times.
- The `benchstat` tool can compare benchmark results.

4.4 Refactoring

You wrote relatively good code so far, and your team is happy. But they think it shouldn't hurt if you could make the code more understandable so that your team can easily maintain it. By refactoring, you can create simpler and more expressive code by reducing complexity and increasing understandability, maintainability, and extensibility.

The tests would guide and tell if you made a mistake. When refactoring, the tests will be on your side because one of the main benefits of testing is that you can change code ruthlessly. The critical thing here is that you shouldn't change tests while changing the code because you can accidentally breed baby bugs. Your goal should be to preserve the behavior of code while refactoring. While doing refactoring, I will help you and explain the strategies you'll use and their reasons.

Refactoring the Parse function

The `Parse` function is an entry point to the url package, and I believe it's the most important function in the url package. People use it to parse a URL and get a parsed `*URL` value. As its name suggests, there is only a single responsibility of the function: Parsing a URL, but is it really like so? Let's discover if this is really true.

Let's take a look at the current `Parse` function in Listing 4.13.

Listing 4.13: The Parse function (url.go)

```
func Parse(rawurl string) (*URL, error) {
    i := strings.Index(rawurl, "://") #A
    if i < 1 {     #A
        return nil, errors.New("missing scheme")          #A
    }   #A
    scheme, rest := rawurl[:i], rawurl[i+3:]    #A
    host, path := rest, ""   #B
    if i := strings.Index(rest, "/"); i >= 0 {  #B
        host, path = rest[:i], rest[i+1:]       #B
    }   #B
    return &URL{scheme, host, path}, nil
}
```

If you look closer, you can see that the `Parse` function has several sub-responsibilities:

1. Detects the position of the scheme signature to parse a scheme.
2. Detects the host position to parse host and path.
3. Returns a new `*URL` with scheme, host, and path.

Things can get messy over time if you don't act now. There can be an endless amount of refactorings, but here's the strategy in my mind: Splitting the sub-responsibilities of the `Parse` function into smaller functions as follows:

1. The first one could be a mini parser function to parse the scheme.
2. The next one could be another mini parser function to parse the host and path.

The `Parse` function will control and call these mini functions to parse a `rawurl`. No worries. You'll soon see how all these things come together throughout this section.

Refactoring the scheme parsing

Let's begin with the first goal: A mini parser function to parse the scheme. To do that, you can follow this plan:

- You can declare an unexported function.
- Move the scheme parsing logic into the function.
- Call the function from the `Parse` function.

Let's do that by moving the scheme parsing logic to a new unexported function in Listing 4.14.

Listing 4.14: Refactoring the scheme parsing (url.go)

```
func Parse(rawurl string) (*URL, error) {
    scheme, rest, ok := parseScheme(rawurl)     #A
    if !ok {       #B
        return nil, errors.New("missing scheme")          #B
    }   #B
    ...
}

func parseScheme(rawurl string) (scheme, rest string, ok bool) {
    i := strings.Index(rawurl, "://")
    if i < 1 {
        return "", "", false #D
    }
    return rawurl[:i], rawurl[i+3:], true
}
```

You should always run tests after you're done with refactoring to be sure you didn't change the code's behavior:

```
$ go test
PASS
```

If you compare the new code in Listing 4.14 to the previous one in Listing 4.13, you can see that Listing 4.14 is now easier to understand than Listing 4.13. Especially because the scheme parsing logic now has a name: `parseScheme`.

Tip

An exported function is like a public function in some other programming languages. On the other hand, an unexported function is similar to a private function. You can export a function by capitalizing its first letter. `ParseScheme` is exported, and other packages can see it. `parseScheme` is unexported, and other packages cannot see it.

In Listing 4.14, you use an *unexported function* called `parseScheme` because other developers don't need to know about the internals of the parsing logic. So there is no need to export the `parseScheme` function.

The `parseScheme` function returns three *named result values*: `scheme`, `rest`, and `ok`. You *named* the return values so that people can see what the function returns without looking at its code. It would be hard to understand what they return if you didn't use named result values:

```
func parseScheme(rawurl string) (string, string, bool) { ... }
```

In Listing 4.14, you also used a *boolean* result value called `ok` that allowed you to report the success and failure of the scheme parsing. You might wonder why it did not return an error instead. This is because the function has two outcomes: *success* and *failure*. This tactic made the caller `Parse` function concise and clear.

You might still ask: Why didn't you return an error from the `Parse` function instead of the `parseScheme` function? This is because the controller is the outer one, the `Parse` function. I think it's better to show what error you would be returning without looking at the inner function: `parseScheme`.

Naked return

The named result values also allow you to return from a function with a naked return:

```
func parseScheme(rawurl string) (scheme, rest string, ok bool) {
    if i < 1 {
        return
    }
    ...
}
```

You don't have to type the result values yourself, and the function will return the current state of the result values. So the code above is equal to the following:

```
if i < 1 {
    return scheme, rest, ok
}
```

But you didn't use a naked return in Listing 4.14 because it often makes code harder to understand. If you had used it, you would have to look at the result values to see what the function was returning, so it's better to return with explicit values.

Refactoring the host and path parsing

Now, you're ready for the next refactoring: Creating a mini parser function to parse the host and path. This refactoring can be similar to the previous one:

- You can declare another unexported parser function.
- Move the host and path parsing logic into the function.
- Call the function from the `Parse` function.

Let's see what it looks like in 4.15.

Listing 4.15: Refactoring the host and path parsing (url.go)

```
func Parse(rawurl string) (*URL, error) {
    ...
    host, path := parseHostPath(rest) #A
    ...
}

func parseHostPath(hostpath string) (host, path string) {
    if i := strings.Index(hostpath, "/"); i >= 0 {
        host, path = hostpath[:i], hostpath[i+1:]
    }
    return host, path
}
```

- You declared a new function called `parseHostPath` to extract the logic for parsing host and path.

- The function returns two result values but doesn't return an error value because the parsing for host and path can never fail.

Let's run the tests:

```
$ go test
PASS
Awesome!
```

Refactoring the common logic

If you look at the mini parser functions in Listing 4.14 and 4.15, you may have noticed that they both almost do the same thing:

- Both of them look for a pattern in a string.
- Return empty strings if they could find the pattern.
- And return some part of the string value if they could.

Let's further refactor these mini parsers. What about creating a common function to make their job easier? You can create another mini function that can search for a pattern in a string and split the string, and return multiple strings. And then, you can use it in the `parseScheme` and `parseHostPath` functions.

There are also `Port` and `Hostname` methods from the previous chapter, and they are similar to the mini parser functions you wrote in this chapter. You can use the same `split` function to refactor them as well.

Let's take a look at the final code in Listing 4.16.

Listing 4.16: Final code after refactoring (url.go)

```
func Parse(rawurl string) (*URL, error) {
    scheme, rest, ok := parseScheme(rawurl)
    if !ok {
        return nil, errors.New("missing scheme")
    }
    host, path := parseHostPath(rest)
    return &URL{scheme, host, path}, nil
}
```

```
func parseScheme(rawurl string) (scheme, rest string, ok bool) {
    return split(rawurl, "://", 1)    #A
}

func parseHostPath(hostpath string) (host, path string) {
    host, path, ok := split(hostpath, "/", 0)   #A
    if !ok {
        host = hostpath
    }
    return host, path
}

// Hostname returns u.Host, stripping any port number if present.
func (u *URL) Hostname() string {
    host, _, ok := split(u.Host, ":", 0)        #A
    if !ok {
        host = u.Host
    }
    return host
}

// Port returns the port part of u.Host, without the leading colo
// If u.Host doesn't contain a port, Port returns an empty string
func (u *URL) Port() string {
    _, port, _ := split(u.Host, ":", 0)         #A
    return port
}

// split s by sep.
//
// split returns empty strings if it couldn't find sep in s at in
func split(s, sep string, n int) (a, b string, ok bool) {
    i := strings.Index(s, sep)        #B
    if i < n {     #C
        return "", "", false #D
    }
    return s[:i], s[i+len(sep):], true #E
}
```

The `split` function searches a pattern in a string value at an index number, and then it returns two distinct string values split by the pattern:

- The first string value is the one before the pattern.
- The second one is right after it.
- It returns empty string values if it couldn't find the pattern in the string at

the given index.
- The boolean result value indicates whether the search was successful.

The final code is kind of verbose, but it's more expressive than the first version in Listing 4.13. It suits my taste buds, but I don't know about you!

Now that you're ready to run the tests:

```
$ go test
PASS
```

Great! It would be hard and error-prone to do this refactoring without the help of the test functions. You should be glad that you have them on your side.

Refactoring a test

Code is not the only thing that you can refactor, and you can also refactor tests itself. It's valuable to make your tests and their output as simple and human-readable as possible. Unlike code that is protected by tests, nothing protects tests. So you have to be super careful and introduce tiny changes while changing tests.

Let's take a look at the current `TestParse` function in Listing 4.17 to understand where you are.

Listing 4.17: TestParse (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://foo.com/go"

    u, err := Parse(rawurl)
    if err != nil {
        t.Fatalf("Parse(%q) err = %q, want nil", rawurl, err)
    }
    if got, want := u.Scheme, "https"; got != want {
        t.Errorf("Parse(%q).Scheme = %q; want %q", rawurl, got, w
    }
    if got, want := u.Host, "foo.com"; got != want {
        t.Errorf("Parse(%q).Host = %q; want %q", rawurl, got, wan
    }
```

```
    if got, want := u.Path, "go"; got != want {
        t.Errorf("Parse(%q).Path = %q; want %q", rawurl, got, wan
    }
}
```

As you can see, the `TestParse` function compares the `*URL` fields one by one and fail with identical error messages, and in these messages, only the field names change. Although doing this is perfectly fine, there is one trick that you can use here. Let's talk about that.

In Go, struct types are *comparable*. So why not just compare `*URL` values instead of comparing fields? You can remove all these field checks when you do that. As always, there is a pitfall: You won't be able to log mismatching field names. You'll see what I mean in a minute.

For example, you can create an expected `*URL` value as follows (as in Listing 4.18):

```
want := &URL{
    Scheme: "https",
    Host:   "foo.com",
    Path:   "go",
}
```

Then you can call the `Parse` function with a raw URL string, get a parsed `*URL` value, and compare it with the expected `*URL` value above, as follows (as in Listing 4.18):

```
got, err := Parse(rawurl)
...
if *got != *want { ... }
```

Tip

You need to use asterisks before the `got` and `want` variables because they are pointers to a `URL` value. For example, the `want` variable above is a memory address that points to a `URL` value in memory. `*want`, on the other hand, is the `URL` value that the `want` pointer points to.

Let's take a look at the refactored test function in Listing 4.18.

Listing 4.18: Refactored TestParse (url_test.go)

```
func TestParse(t *testing.T) {
    const rawurl = "https://host/some/fake/path"          #A

    want := &URL{            #B
        Scheme: "https",     #B
        Host:   "foo.com",   #B
        Path:   "go",        #B
    }   #B

    got, err := Parse(rawurl)         #C
    if err != nil {
        t.Fatalf("Parse(%q) err = %q, want nil", rawurl, err)
    }
    if *got != *want {       #D
        t.Errorf("Parse(%q):\n\tgot:  %q\n\twant: %q\n", rawurl,
    }
}
```

Listing 4.18:

- Creates a fake rawurl so you can see the failure message when you run the test function.
- Then it creates an expected `*URL` value for "`https://foo.com/go`".
- Runs the `Parse` function and gets a new parsed URL.
- Finally, it compares the expected and wanted URLs.

The test will print a descriptive error message as follows if the URL values do not match:

```
$ go test -run TestParse
--- FAIL: TestParse
    Parse("https://host/some/fake/path"):
        got:  https://host/some/fake/path
        want: "https://foo.com/go"
```

Tip

The `Errorf` method in Listing 4.18 automatically calls the `String` method on each `*URL` value to print them inside double-quotes whenever the method sees the "`%q`" verb.

I liked this failure message because you can clearly see the expected and wanted values. But it is kind of hard to see which fields are mismatching. Wouldn't it be better if you could see which fields are mismatching without losing the convenience of directly comparing struct values?

One solution is printing the URL values using the "`%#v`" verb instead of the "`%q`" verb as follows:

```
t.Errorf("Parse(%q):\n\tgot  %#v\n\twant %#v\n", rawurl, got, wan
```

And the output looks like this:

```
got:  &url.URL{Scheme:"https", Host:"host", Path:"some/fake/path"
want: &url.URL{Scheme:"https", Host:"foo.com", Path:"go"}
```

Remember

The `%q` verb wraps a string in double-quotes. The `%#v` verb formats a value in the Go syntax. You can find all the other verbs at the link: https://pkg.go.dev/fmt.

This output is better, but it's kind of verbose, and it can be hard to read if you were to compare a lot of URLs in the future.

There is one more trick in my bag of tricks: Using a helper method that I call `testString`. You can put it next to the `String` method and use it to return a simple string only for tests (Listing 4.19).

Listing 4.19: Adding the testString method to the URL type (url.go)

```
func (u *URL) String() string { ... } #A

func (u *URL) testString() string {
    return fmt.Sprintf("scheme=%q, host=%q, path=%q", u.Scheme, u
}
```

A new method in the `URL` type called the `testString` returns a concise representation of a `URL` value. This method could be on the testing side but keeping it near the `String` method is more practical. For example, when you want to change the `String` method, you can easily see what to change in the `testString` method. You can put it into a test file as a stand-alone function if you like it that way. But then, you would lose the proximity benefit.

You can now use it in the failure message as follows:

```
t.Errorf("Parse(%q):\n\tgot  %s\n\twant %s\n", rawurl, got.testSt
```

Since the `testString` method would return a string, the `Errorf` method uses the "`%s`" verbs instead of "`%v`" verbs. The output should look like the following when you run the test:

```
$ go test -run TestParse
--- FAIL: TestParse
    Parse("https://host/some/fake/path"):
        got:  scheme="https", host="host", path="some/fake/path"
        want: scheme="https", host="foo.com", path="go"
```

The failure message is now more evident to my developer's eyes as it is now straightforward to find which fields are not matching. So is the `testString` method worth the hassle? I think so because the `testString` method is more versatile than the previous solutions, and it can also help you find bugs when you're debugging, so it's not just for tests.

About go-cmp

A third-party package called `go-cmp` allows you to compare complex types with each other and show their differences in a straightforward way.

First, you need to add it as a module as follows:

```
go get -u github.com/google/go-cmp/cmp
```

Then import it in a test file and use it to compare values like this:

```
import "github.com/google/go-cmp/cmp"
...
func TestParse(t *testing.T) {
    ...
    if diff := cmp.Diff(want, got); diff != "" {
        t.Errorf("Parse(%q) mismatch (-want +got):\n%s", rawurl,
    }
}
```

The failure message will look like this:

```
--- FAIL: TestParse
    Parse("https://host/some/fake/path") mismatch (-want +got):
          &url.URL{
              Scheme: "https",
              -   Host:   "foo.com",
              +   Host:   "host",
              -   Path:   "go",
              +   Path:   "some/fake/path",
          }
```

It would be unnecessary to use it in the url package because the `URL` type doesn't contain complex fields. So the `go-cmp` package is more helpful in comparing types that have pointers, slices, etc.

Wrap up

- Refactoring helps you create maintainable code.
- Your goal should be to preserve the behavior of code while refactoring.
- You shouldn't change tests while refactoring; otherwise, you can introduce bugs.
- Unlike code that is protected by tests, nothing protects tests. So you have to be super careful and introduce tiny changes while changing tests.

4.5 External tests

Before finishing the chapter, let's talk about external tests. In section 4.1, you created example functions for the url package in an external package called `url_test`. You may be wondering about the differences between external and internal tests.

Should you write an external test or an internal test? You may not be sure which one to use in what situation. It's time to explain the differences between external and internal tests and when to use which one.

You'll also learn a trick for testing the unexported part of your code, even if you're using an external test. Don't worry. You'll understand what I mean soon.

4.5.1 Internal vs. external tests

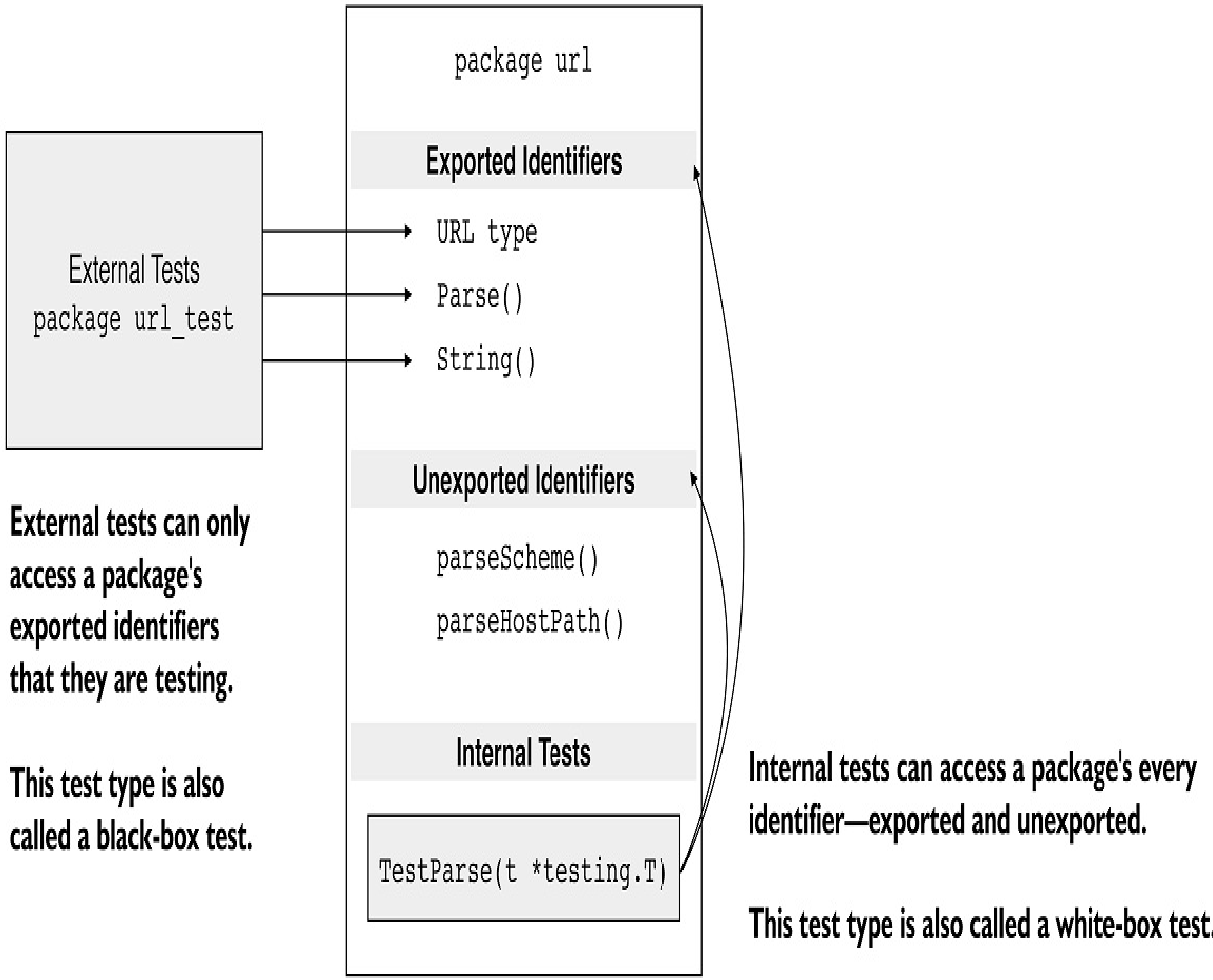
Let's talk a bit about external vs. internal tests:

- An internal test is a test that verifies code from the same package. This test type is also called a *white-box* test.
- An external test, on the other hand, is a test that verifies code from another package. This type of test is also called a *black-box* test.

Just like internal tests, you put external tests in the same folder with the code you test. Usually, there cannot be multiple packages in the same folder. But as an exception, an external test package can live alongside the package that it verifies.

You can see what an external and internal test look like in Figure 4.8.

Figure 4.8 External vs. internal tests



- You can see that the `url` package has exported and unexported identifiers.
- It also contains an internal test called `TestParse`.

External tests can only access the exported identifiers of a package that they're testing. In Figure 4.8, you can see that the `url_test` package can only access the exported identifiers: The `URL` type, `Parse` and `String` functions. Since they cannot see the internals of the url package, all the tests in the external `url_test` package are called *black-box tests*.

Internal tests can access all the identifiers of a package that they're testing, no matter whether the identifiers are exported or not. That's why they are called *white-box tests*. In Figure 4.8, you can see that the `TestParse` function can access all the identifiers of the `url` package: The `URL` type, `Parse`, `String`, `parseScheme`, and `parseHostPath` functions.

Benefits of external tests

When you write an external test, you'll be externally testing your code, and you can only see the exported identifiers and cannot access the unexported ones. The benefit of using an external test is that you can only test the visible behavior of your code, and your tests can only break if the API of your code changes. Of course, these tests can also break if there's a bug.

Another advantage of external tests is that they can prevent possible import cycles. On the other hand, with an internal test, if you don't have the discipline, your test code can be fragile and break when the code you're testing changes.

4.5.2 Testing unexported identifiers

Often you might *not* want to test the internals of a package. As I explained in the previous heading, testing the internals of a package may make your tests brittle. But sometimes, it may be necessary to do so. So let's talk about your options when you want to test the internal parts of your package.

Let's say you and your team decided to write external tests instead of internal tests. For example, suppose for some odd reason you want to test the unexported `parseScheme` function you wrote earlier:

```
package url
func parseScheme(rawurl string) (scheme, rest string, ok bool) {
    ...
}
```

You could easily test the `parseScheme` function by creating another test in the `url_test.go` file. You can do so because the test file is internal and belongs to the same `url` package with the `parseScheme` function:

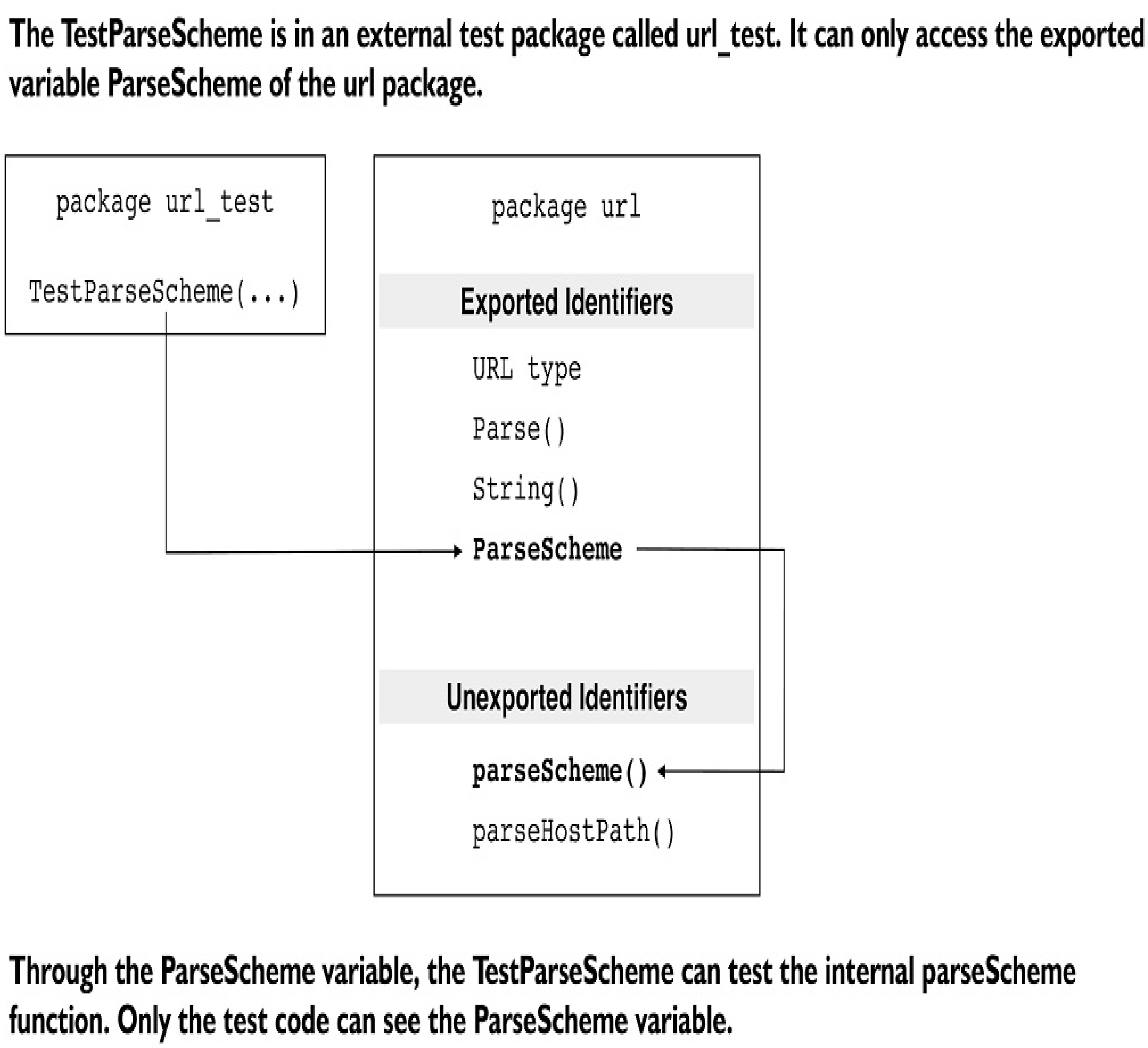
```
package url
func TestParseScheme(t *testing.T) {
    scheme, rest, ok := parseScheme(rawurl)
    ...
```

```
}
```

But since you decided to use an external test, you can't easily test the `parseScheme` function. It's because the function is unexported from the `url` package. Fortunately, there is a helpful trick that allows you to test an unexported function. All you need to do is to export the function from the `url_test` package instead!

Functions are first-class citizens in Go, so you can assign a function to a variable and call the variable as a function later on (Figure 4.9).

Figure 4.9 The external test is testing the unexported function



- The `TestParseScheme` is in an external test package called `url_test`.
- The `TestParseScheme` can only access the exported variable `ParseScheme` of the `url` package, but it cannot directly access the unexported `parseScheme` function.
- Through the `ParseScheme` variable, the `TestParseScheme` can test the internal `parseScheme` function.
- Only your test code can see the `ParseScheme` variable. Since the `ParseScheme` variable is in a test file, the test tool will compile it alongside the url package. However, the Go compiler won't add it to the final binary. So, other packages cannot see it but your test.

I think it would be unfair if I don't show you how to implement this diagram in code. Let's do that! In Listing 4.20, you'll create a new file called "export_test.go", export an unexported function, then export it using an exported variable. Sounds complicated? No worries. I'll explain what's going on in a second.

Listing 4.20: Exporting a function for testing (export_test.go)

```
package url
var ParseScheme = parseScheme          #A
```

As you can see in Listing 4.20:

- You exported the `parseScheme` function from the `url` package.
- Since the `ParseScheme` variable is in a test file, the test tool will compile it alongside the `url` package.
- So you can access the variable in our test files as if it was being exported from the `url` package!

You have an exported variable called `ParseScheme`, so you can now access the unexported `parseScheme` function. It's time to create a new test function to test the `parseScheme` function using this exported variable called `ParseScheme`.

Let's create a new test file called "`parse_scheme_test.go`", and write a test called `TestParseScheme` in Listing 4.21.

Listing 4.21: Testing the parseScheme (parse_scheme_test.go)

```
package url_test   #A

import (
    "testing"
    "github.com/inancgumus/effective-go/ch04/url"
)

func TestParseScheme(t *testing.T) {
    const (
        rawurl = "https://foo.com/go"
        wantScheme = "https"
        wantRest = "foo.com/go"
        wantOk = true
    )
    scheme, rest, ok := url.ParseScheme(rawurl) #C
    if scheme != wantScheme {
        t.Errorf("parseScheme(%q) scheme = %q, want %q", rawurl,
    }
    if rest != wantRest {
        t.Errorf("parseScheme(%q) rest = %q, want %q", rawurl, re
    }
    if ok != wantOk {
        t.Errorf("parseScheme(%q) ok = %t, want %t", rawurl, ok,
    }
}
```

- The code in Listing 4.21 is in a new external test file that tests the `url` package.
- Since the test is an external package, you need to import the `url` package.
- Then you tested the `parseScheme` function through the exported `ParseScheme` function.

You may be wondering why you hustled so much. You might ask why you didn't export the `parseScheme` function in the "`url.go`" file instead?

If you did so, you'd be exposing the function to other developers, and you don't want to do that because that function is an internal part of the `url` package. You might want to change the function in the future without breaking the importers of the package.

With the code in Listing 4.21, you exposed the function only for the tests! Notice that the code is in a test file called `export_test.go`. So it's a test file, and the Go compiler will ignore it when you want to build the package. So other developers won't be able to access the `ParseScheme` variable when they import the `url` package.

4.5.3 Wrap up

I often prefer writing internal tests and avoid the work that comes with external tests. With an internal test, you don't need to import the package that you are testing. As always, choosing between external and internal tests depends on you. Pick your poison.

Let's summarize what you've learned:

- External tests reside in a package with a `_test` suffix and cannot access the unexported identifiers of a package. But they can verify the internals of a package using a trick to export an unexported identifier via an exported one. On the other hand, internal tests reside in the same package as the code they test, and they can access both exported and unexported identifiers from the package.
- Internal tests are white-box tests and verify code from the same package.
- External tests are black-box tests and verify code from an external test package.
- One benefit of external tests is that they verify the visible behavior of code, and these tests can only break if the API of the code changes (unless there is a bug).

4.6 Summary

Phew! That was a long chapter. Well done if you read it this far! The url package is now ready: Documented, has total test coverage, benchmarked, and refactored.

Let's see what you've learned in this chapter:

- Testable examples allow you to create documentation that never goes out of date.

- Test coverage helps you find untested code areas, but it doesn't guarantee 100% bug-free code.
- You can measure code performance using benchmarks.
- Refactoring helps you create maintainable code, and tests help you to refactor your code without fear.
- External tests allow you to write black-box tests and test the public area of a package. Internal tests allow you to write white-box tests and test every aspect of a package.

5 Writing a Command-Line Tool

This chapter covers

- Writing and testing a CLI tool.
- Parsing and validating command-line arguments and flags.
- Extending the flag package with custom types.

A long time ago, alone at home, I was typing on the command line:

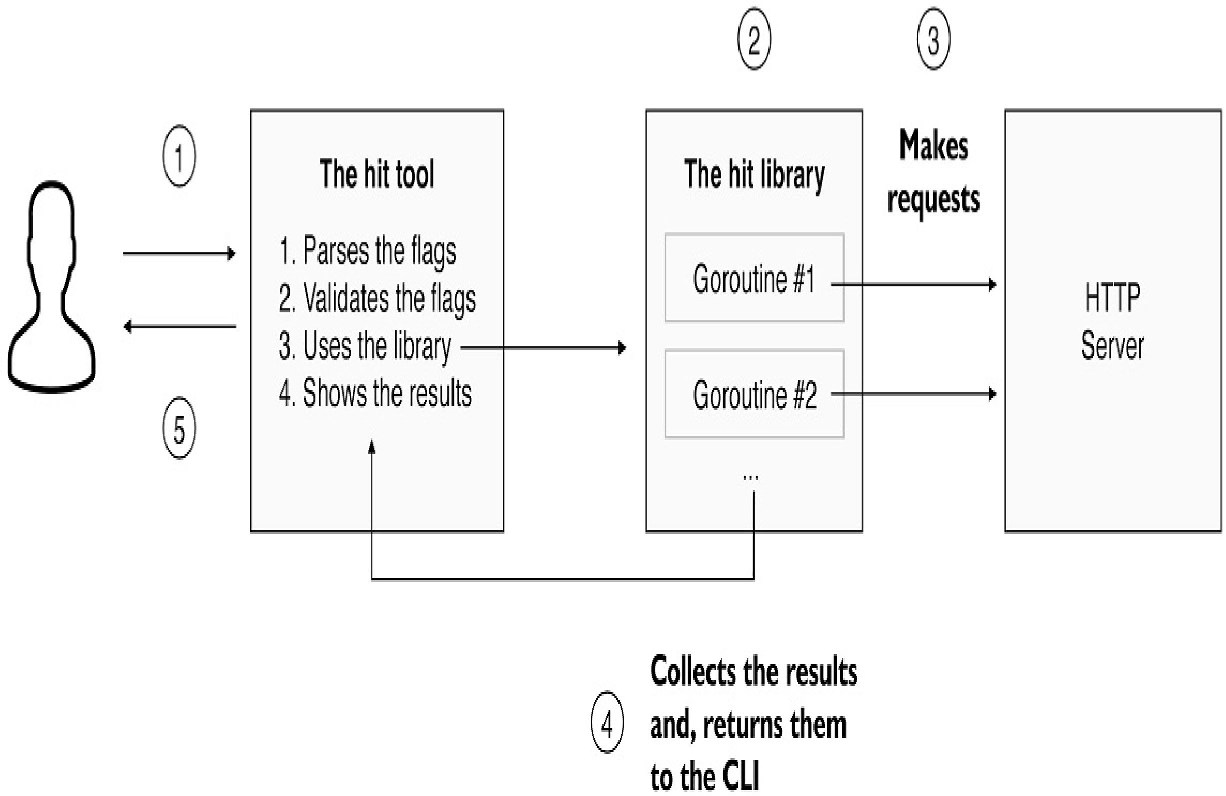
```
C:\> dir
  Volume in Drive C is MS_DOS_5
  . . .
C:\> A:
A:\> PRINCE.EXE
<<Prince of Persia video game starts>>
```

Since then, things have changed, but the command line still persists. Some people often prefer to work on the command line to get their tasks done quickly. Some command-line tools launch a web server, and others search for files or run your tests.

Imagine your team wants to make a command-line interface—*as known as a CLI*—tool called "*hit*" in Go to make concurrent requests to an HTTP server and gather performance metrics. You know the basics, but you don't know how to create and test a command-line tool in Go. In this chapter, I'll teach you to write an idiomatic, maintainable, and testable command-line tool from scratch using the Go Standard Library.

As you can see in Figure 5.1, a user runs the hit tool with a few command-line arguments (*not shown in the figure*) to check how an HTTP server performs. The tool makes the requests to the HTTP server, aggregates response statistics, and shows the result to the user.

Figure 5.1 The hit tool's bird's-eye view architecture.



1. A user runs the *hit tool* (tool for short).
2. The tool uses *the hit library* to make HTTP requests.
3. The hit library package fires up goroutines and makes several requests to a server.
4. Then the hit library package gathers the results and returns them to the tool.
5. Finally, the tool shows the results to the user.

In this chapter, you'll write and test the tool from scratch. First, you will learn how to organize the tool's directories, packages, and files. You will display a usage message and cross-compile the initial version of the tool for different operating systems.

After that, you will learn about parsing command-line arguments to let users change the tool's behavior. Users do not always pass proper arguments to a tool, so you'll also learn how to validate command-line arguments.

The Go Standard Library offers a great package for parsing command-line arguments. However, sometimes it falls short because it cannot provide everything built-in for special cases. You will learn how to extend and customize it to make it more powerful. Finally, you will test the tool by separating and abstracting some parts of it.

After you finish writing the hit command-line tool in this chapter, in the next chapter, you'll write and integrate the hit library package to the hit tool and start making concurrent requests to an HTTP server.

Alright, let's get started!

5.1 Getting a preview

As I explained in the chapter entry, you will build a new command-line tool that will make concurrent requests to an HTTP server, collect the results, and show it to a user. You learned how it will work in a bird's eye view and what packages it will use. Let's discuss how you will interact with the tool from the command line and which flags you will be passing to it. So you can better understand what you will be creating.

Suppose you want to make 100 requests to the local server's 9090 port with a concurrency level of 10. You will run the hit tool as follows to do that:

```
$ hit -url=http://localhost:9090/ -n=100 -c=10
```

The tool will display a banner and show where to make requests (`https://localhost:9090`), how many requests it will make (`100`), and with what concurrency level (`10`) (1st step in Figure 5.1):

```
__  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Making 100 requests to http://localhost:9090 with a concurrency l
```

- The hit tool will call the hit library (2nd step in Figure 5.1).
- The hit library will make 100 requests to the server by distributing the

requests among 10 concurrent goroutines (3rd step in Figure 5.1).
- Then it will collect the response statistics, aggregate them, and return the aggregated results to the hit tool (4th step in Figure 5.1).

Finally, the hit tool will print them as follows (5th step in Figure 5.1):

```
Summary:
  Success   : 100%
  RPS       : 3.45
  Responses : 100
  Errors    : 0
  Duration  : 29s
  Fastest   : 1s
  Slowest   : 4s
```

That's all. Short and sweet.

Arguments vs. named arguments vs. flags

Go calls command-line arguments *flags*, and you will see me calling them like so in the rest of the chapter. I will call them *arguments* when referring to the program's raw input from the command line.

5.2 Writing your first tool

The previous section showed you that you will be creating a command line tool called "hit", what packages it will use and their interactions, and how you will interact with it from the command line.

This section will first show you how to organize the hit tool's packages and create the necessary directories and files. Organizing your code is important because it helps to keep code simple and maintainable.

After learning about code organization and creating the basic structure, you'll learn how to display a usage message to users. A usage message usually includes information about changing a tool's behavior. For example, the hit tool will let users decide which server to make requests to.

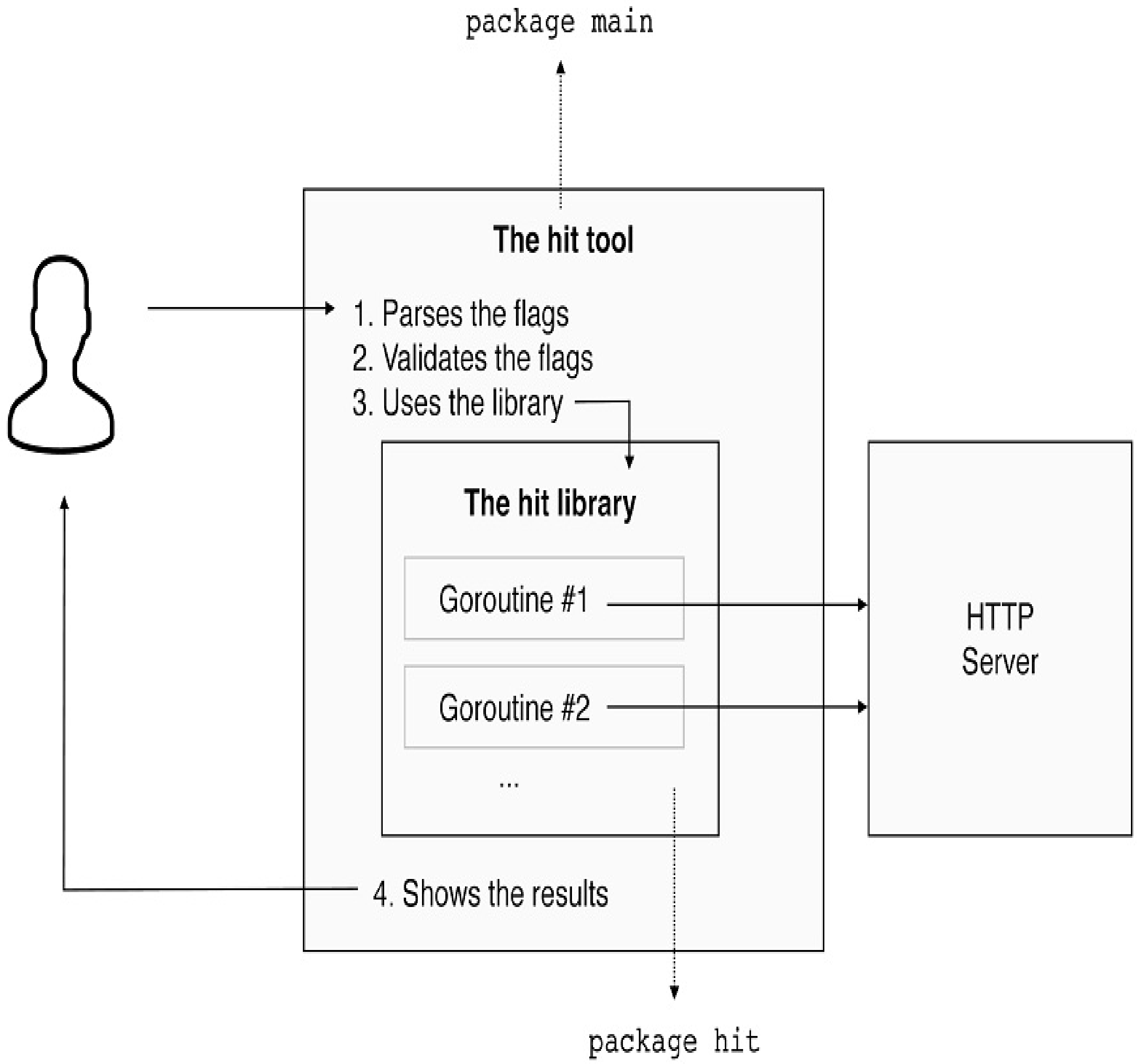
Lastly, you'll learn how to cross-compile the tool so that you can write the tool once and run it on different operating systems.

Let's create your first tool!

5.2.1 Code organization

Like I said in the section introduction, organizing your code is important because it helps to keep code simple and maintainable. This subsection will show you what directories and files to create for the hit command-line tool. As shown in Figure 5.2, there will be two packages—hence two directories because each package should be in a separate directory in Go.

Figure 5.2 The package structure of the hit tool.



1. *The hit tool*—The tool will be in the `main` package, and you will create it in this chapter. It will provide a command-line interface to users.
2. *The hit library*—The library will be in the `hit` package, and you will create it in the next chapter. It will contain the logic of making HTTP requests. The hit command-line tool will import this package to make requests.

Warning

About Go Modules

If you have downloaded the code from the book's github repository, you would already have the code for the tool in the ch05 directory.

If you want to code along with the book and write code from scratch, you might wish first to create a new directory and initialize a new Go module.

First, go to your home directory (Linux/macOS):

```
$ cd ~
```

On Windows:

```
C:\> cd %HOMEPATH%
```

Then type the following commands:

```
$ mkdir project_name
$ cd project_name
```

Type the following to initialize a new Go module while in the root directory:

```
$ go mod init github.com/your_username/project_name
```

Let's create the following directories under a new directory (depicted with a *dot* below):

```
.                        -> Current directory
└── cmd                  -> A directory for executables
│   └── hit              -> The hit CLI tool's directory
└── hit                  -> The hit library's directory
```

The `./cmd/hit/` directory will contain the hit tool, and the `./hit/` directory will contain the hit library. Organizing a CLI tool in this way makes it manageable and reusable! For example, you can create an HTTP server, put it in a new directory under the `cmd` directory, and call the hit library. Or, you can let other people import the hit library package and use it in their own programs.

Tip

The hit tool and library are in two separate packages. This separation of concerns approach makes you create maintainable and testable command-line tools.

It's time to create the first file—which will be the entry point to the program:

1. Make sure that you're in the `/cmd/hit` directory.
2. Create a file called `hit.go`.
3. Open the file in your favorite editor, and you're good to go!

Note

There are two competing conventions: Naming the entry point file "`main.go`" or naming the entry point with the same name as the package (in this case "`hit.go`"). Although most gophers usually create a file named `"main.go"` as an entry point to their programs, it's up to you. If you want to follow the first convention, just name it `main`. Calling it `main` may hint to other developers that that file is the entry point, and they should start reading from that file.

Go does not dictate a package or directory structure

What you should be worrying more about are package names. A package name should tell you and others what it provides and be unique. A unique package name can help its importers to distinguish it from other packages.

Don't worry too much about package organization or directory structure from day one, and wait for it to reveal itself in time. Keep everything as simple as possible.

See the links for more information:

- https://go.dev/blog/package-names

- https://go.dev/blog/organizing-go-code

5.2.2 Printing a usage message

In Go, when you want to make a command-line tool, you add the `main` package and `main` function. Doing so will mark the `main` function as an entry point to your program so that you can execute the program from the command-line. Let's make things more interesting and add the first function, shall we?

In this section, you will create a single function called `main` in the `main` package. And, the `main` function will be an entry point for the hit tool. When users execute the tool, they will see a fancy banner and know they run the correct tool (also to impress them!). And, they will see a usage message and get to know how to use the tool.

Listing 5.1: The first version (hit.go)

```
package main       #A

import "fmt"

var (

    bannerText = ` #B
 __  __      __      ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/
`[1:]   #C

    usageText = `  #B
Usage:
  -url
       HTTP server URL to make requests (required)
  -n
       Number of requests to make
  -c
       Concurrency level`[1:]         #C
)

func main() {      #A
    fmt.Println(bannerText)
    fmt.Println(usageText)
}
```

Note

You use two separate variables because you'll be using them at different times later.

The `package main` and the `func main` always go together. The Go linker will arrange the `main` function as the entry point to the program. `[1:]` removes the first newline character from the texts. You could have typed the *raw string literal* without a leading new line, but it would look skewed in the source code.

Beware

Do not add other characters than the new line after the first backquote (`` ` ``) character when declaring the variables in Listing 5.1. Otherwise, the Go compiler will also include those characters you added in the variables, and `[1:]` will only remove the first character, instead of the newline character.

Make sure that you're under the `./cmd/hit` directory and run the program as follows:

```
$ go run .
 __  __     __     ______
/\ \_\ \   /\ \   /\__  _\
\ \  __ \  \ \ \  \/_/\ \/
 \ \_\ \_\  \ \_\    \ \_\
  \/_/\/_/   \/_/     \/_/

Usage:
  -url
       HTTP server URL to make requests (required)
  -n
       Number of requests to make
  -c
       Concurrency level
```

Looks gorgeous. Now, this is fun!

Raw string literals

The Go compiler does not interpret what is in a raw string literal. You can use a raw string literal to create a multi-line string value, and you don't have to use escape sequences such as `\n` (newline).

The `bannerText` variable in Listing 5.1 uses a *raw* string literal.

If it were to use a string literal, it would look as follows:

```
var bannerText = " __  __     __     ______"+
    "/\ \_\ \   /\ \   /\__  _\"+
    ...
```

Looking weird, and it has an error. You would need to type `\\` instead of a single `\` as follows (The Go compiler interprets what is in a string literal and a backslash is an escape sequence):

```
"/\\ \\_\\ \\   /\\ \\   /\\__  _\\"+
```

Looking weirder.

Beware of global variables

Do you see a problem with the code in Listing 5.1? The `bannerText` and `usageText` are package-level variables accessible throughout the main package, and they may cause trouble down the line if you're not careful.

Tip

Having package-level variables might not be your best idea. Take a look at the link for a discussion about the global state: https://softwareengineering.stackexchange.com/questions/148108/why-is-global-state-so-evil.

What to do? You can easily make them read-only by using constants. Then you wouldn't be able to trim the first newline character. Listing 5.2 solves the slicing problem using functions.

Listing 5.2: The final version (hit.go)

```
...
const ( #A
    bannerText = `...`       // cannot use [1:]
    usageText = `...`        // cannot use [1:]
)
```

```
func banner() string { return bannerText[1:] }  #B
func usage() string  { return usageText[1:]  }  #B

func main() {
    fmt.Println(banner())    #C
    fmt.Println(usage())     #C
}
```

Listing 5.2 uses constants, and nobody can change them (except you—as the code's author). Then you use simple functions to trim the first newline characters from the constants. Finally, you call the functions in the `main` function to print the banner and usage message.

Tip

If you worry about creating a lot of string values each time you call the `banner` or `usage` functions, don't worry. Trust the Go compiler. It is bright enough that it won't allocate a new string value on memory each time you call these functions. Instead, it will embed the string values once into the final executable. Even the slice expression would be simple pointer arithmetic to the same string values.

Why does Go not allow slicing constants?

```
const bannerText = `...`[1:]
```

Are you wondering why the code above won't compile? The answer is simple enough: The Go specification won't allow it. You can read this post from Robert Griesemer if you're still wondering why: https://groups.google.com/g/golang-dev/c/uzBxK5FKV1c/m/XIIXT-b1CwAJ

Before reading the answer, you need to learn a few things about constants in Go, though:

· They can be both *typed* and *untyped*.

· You can read my article at the link to learn more: https://blog.learngoprogramming.com/learn-golang-typed-untyped-constants-70b4df443b61

· Make sure to read the blog post by Rob Pike at the link too: https://go.dev/blog/constants

5.2.3 Cross-compiling the tool

You created the hit tool's directories and packages. Then you created the main function in the main package and made an executable program. The hit tool can print a banner and usage message to its users.

But, think about it, does everybody use macOS? Or Linux or Windows? Of course, not. Everyone has their favorite operating system. So, you might want to cross-compile the hit tool for different operating systems if you want to expand your user base and help more people.

It's time to learn how to cross-compile the tool for various operating systems. Everything starts with the first step. So, let's first learn how you can compile the tool for your operating system.

Go to the "`cmd/hit`" directory and type the following:

```
$ go build
```

This command will create an executable named `hit` (or `hit.exe` if you're on Windows) in the same directory. Now you can run it as follows:

```
$ ./hit
```

For Windows users out there, you can run it as follows:

```
hit.exe
```

GOOS and GOARCH

So far, so good. Let's now learn how to compile the tool for other operating systems than the one you're running. There are two environment variables that you can use to compile your program for a specific operating system and architecture:

- `GOOS` (pronounced "goose") stands for "Go Operating System".
- `GOARCH` (pronounced gore-ch) stands for "Go Architecture".

You can see what your machine is currently using by typing the following command:

```
$ go env GOOS GOARCH
Darwin
arm64
```

Run the following command if you want to see the list of all the operating systems that you can compile:

```
$ go tool dist list -json
```

Suppose that you're using the macOS operating system and want to compile for Windows. You can do that as follows:

```
$ GOOS=windows GOARCH=amd64 go build
```

This command will create a file named "hit.exe" in the same directory. You can change the directory and binary name to something else if you want:

```
$ GOOS=windows GOARCH=amd64 go build -o bin/hit.exe
```

This last one will create a `bin` directory (if it does not exist) and add a file named `hit.exe` in it.

Using a Makefile

Finally, let's say you want to compile the tool for multiple operating systems. Doing so can be a hassle every time you want to make a release. So, you might want to make things easier for you. For example, you can create a `Makefile` and compile the tool for all operating systems in one go.

Beware

You need to install the `make` tool first. For Windows, take a look at the link: https://stackoverflow.com/questions/32127524/how-to-install-and-use-make-

in-windows

While you're in the `cmd/hit` directory, go two directories above (the root directory of the tool). Then create a file named `Makefile` and type the following in it:

```
compile:
	# compile it for Linux
	GOOS=linux GOARCH=amd64 go build -o ./bin/hit_linux_amd64
	# compile it for macOS
	GOOS=darwin GOARCH=amd64 go build -o ./bin/hit_darwin_amd
	# compile it for Apple M1
	GOOS=darwin GOARCH=arm64 go build -o ./bin/hit_darwin_arm
	# compile it for Windows
	GOOS=windows GOARCH=amd64 go build -o ./bin/hit_win_amd64
```

Beware

Make sure you're using tabs instead of spaces at the beginning of a line in the Makefile (vital there, doesn't matter elsewhere).

You can now run it as follows:

```
$ make
```

Doing so will compile the tool for each operating system in the `Makefile` and create the executables in the `bin` directory:

```
hit_darwin_amd64
hit_darwin_arm64
hit_linux_amd64
hit_win_amd64.exe
```

Happy cross-compiling!

5.2.4 Wrap up

This section taught you how to write your first command-line tool called hit. It has a maintainable directory and package structure, and it can print a banner and usage message. These were good first steps.

Let's talk a bit about what you have learned in detail:

- The hit tool is a command-line tool to make concurrent requests to an HTTP server to measure the server's performance.
- An executable program needs to be in the package main with the main function. The main function is an entry point to the program, and it gets executed when you run your program.
- The cmd directory helps you organize the executable packages. You can put another directory under the cmd directory for each of your executables.
- Separating the CLI from the logic allows you to create reusable and testable code.
- You can cross-compile by using `GOOS` and `GOARCH` environment variables.

5.3 Parsing command-line flags

You wrote the first part of the command-line tool, but it's not pretty useful yet. What can you improve here? Well, users can change the hit tool's behavior using command-line flags to decide how many requests to make and where.

Let's remember what the flags look like:

```
Usage:
  -url
       HTTP server URL to make requests (required)
  -n
       Number of requests to make
  -c
       Concurrency level
```

The `url` flag will let users choose which server to make HTTP requests. The `n` flag will let them choose how many requests to make to the server. And lastly, the `c` flag will let them adjust the concurrency level. It's time to learn how to get and parse these flags from the command-line.

You have two options when it comes to getting and parsing command-line flags:

1. Getting them directly using a lower-level package called `os`.
2. Getting them by using a higher-level package called `flag`.

The first one is simpler and gets you raw data. The latter provides an abstraction on top of the `os` package and provides a structured and flexible way of getting and parsing command-line flags.

1. First, you will create the basic skeleton for flag parsing.
2. Then you'll make a flag parser using the `os` package.
3. Lastly, you'll upgrade the parser to the `flag` package.

Let's get started!

5.3.1 Storing and parsing flags

In this section, you'll create the barebones of a new type for storing and parsing flags. The parser won't parse anything in its initial form. However, it will help you add the parser code later in this section.

Let's create a new type in a new file called `flags.go` in the `./cmd/hit` directory (Listing 5.3). Doing so will prevent you from polluting the `main` function with the parsing logic.

Listing 5.3: Adding a flag parser (flags.go)

```
package main

type flags struct {          #A
    url  string              #A
    n, c int                 #A
}       #A

func (f *flags) parse() error {                    #B
   // ...parsing code will be here                 #B
   return nil                                      #B
}                                         #B
```

Listing 5.3 adds a new struct type called `flags` for storing the default and parsed command-line flag values.

- The `url` field is the URL to make HTTP requests.
- The `n` field is the number of requests.
- The `c` field is the concurrency level.

The `parse` method will parse the command-line arguments and return an error if it cannot parse them. Next, let's integrate the flag parser into the `main` function (Listing 5.4).

Listing 5.4: Integrating the parser (hit.go)

```
...
import (
    "fmt"
    "log"
    "runtime"
)
...
func main() {
    var f flags
    if err := f.parse(); err != nil {
        fmt.Println(usage())          #A
        log.Fatal(err)                          #A
    }
    fmt.Println(banner())
    fmt.Printf("Making %d requests to %s with a concurrency level
       f.n, f.url, f.c)
}
```

Note

The `Fatal` function prints an error and ends the program with an exit code of one to the operating system.

The `main` function in Listing 5.4 calls the `parse` method to parse the flags. If the parsing fails, it prints a usage message and terminates the program using the `Fatal` function. Let's try it (while in the `cmd/hit` directory):

```
$ go run .
...
Making 0 requests to  with a concurrency level of 0.
```

Of course, the current output prints the zero-values of the fields. Then again, it's party time because the bare structure of the tool is ready!

5.3.2 The os package

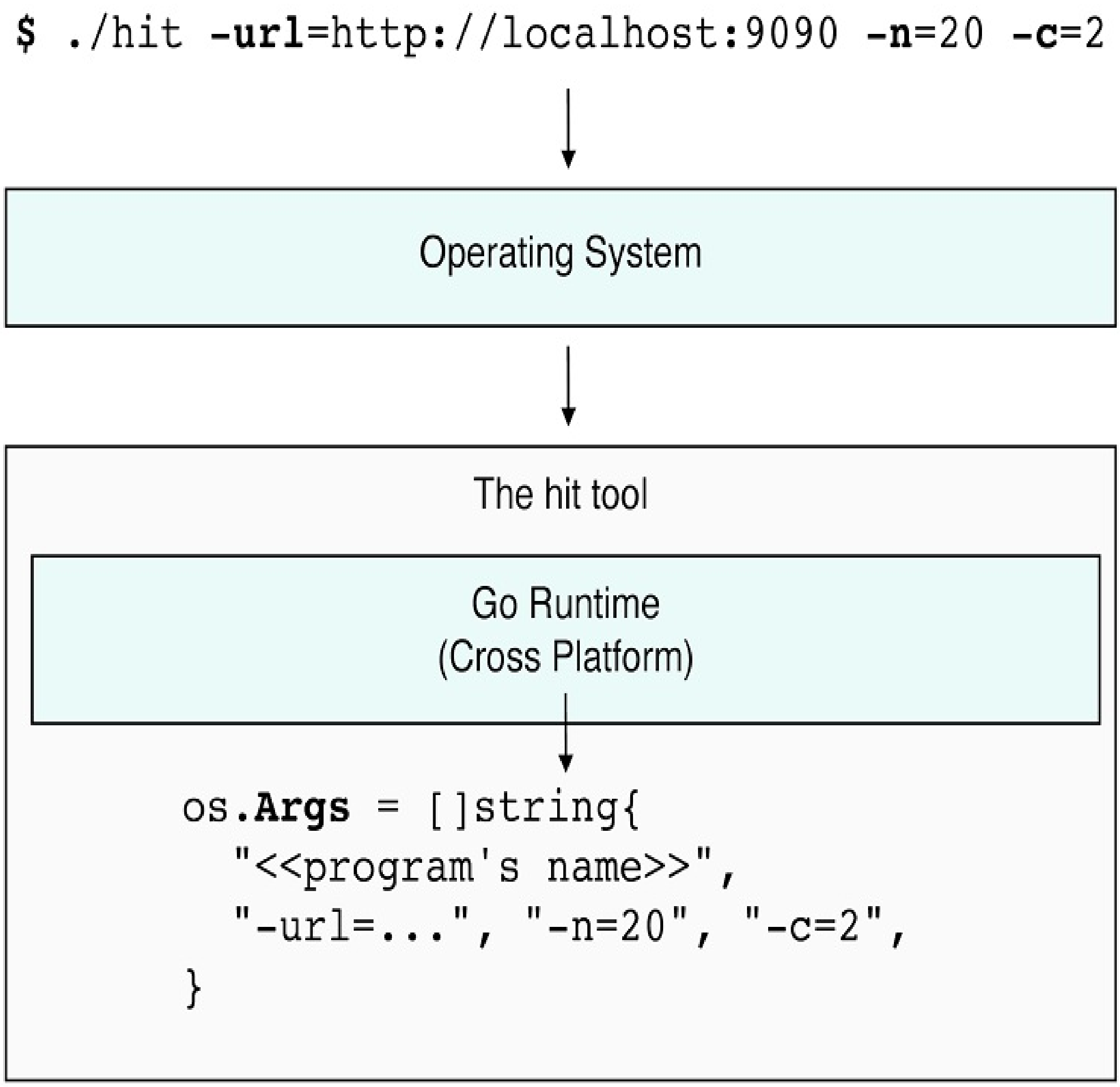
The `os` package is one of the options you have for getting command-line flags in a cross-platform way. It abstracts the underlying operating system so that you can create cross-platform programs. Let's say you compiled your program to run on Linux. The `os` package's programming API will stay the same for you, but it will have the necessary code to talk to the Linux operating system.

By default, the `flag` package uses the `os` package. So it might be a good idea to lift the curtains a little bit and look at the `os` package first.

Getting command-line arguments

You will get and parse three arguments: URL, number of requests, and concurrency level. Let's talk about how you can get them from the command line in Figure 5.3.

Figure 5.3 The Go runtime fills the Args variable.



In Figure 5.3:

1. A user runs the hit tool with three flags.
2. The operating system passes the command-line arguments to the program.
3. The Go runtime gets the arguments from the operating system and initializes the os package's `Args` variable as a slice of string.

The first element in the slice is the program's name, and the rest are the command-line arguments. You're often interested in the command-line arguments, not the program name. So, you can skip the program name as follows: `os.Args[1:]`.

Tip

Getting the arguments from an operating system needs to be cross-platform because each platform has its own requirements, such as different pointer sizes. Search for the `sysargs` function in the Go source code if you're curious. Let me leave a link for you here: https://github.com/golang/go/search?q=sysargs.

Implementing a command-line flag parser

You implemented a parser in the last part of the book, and you don't need to learn how to make another one. Then again, it wouldn't hurt to look at an example implementation of a parser that can parse command-line flags using the `Args` variable. So, you can learn how to parse command-line flags yourself. This knowledge will prepare you for understanding the flag package.

Beware

The Cut function in Listing 5.5 comes with Go 1.18. Earlier Go versions don't have it. You can download the latest version of Go at the link: https://go.dev/dl/. You can see the installed Go version on your machine by typing the following at the command-line: `go version`.

Listing 5.5: Parsing using os.Args (flags.go)

```
...
// parseFunc is a command-line flag parser function.
type parseFunc func(string) error     #A

func (f *flags) parse() (err error) {
    // a map of flag names and parsers.
    parsers := map[string]parseFunc{  #B
        "url": f.urlVar(&f.url), // parses an url flag and update
        "n":   f.intVar(&f.n),   // parses an int flag and update
        "c":   f.intVar(&f.c),   // parses an int flag and update
```

```
    }
    for _, arg := range os.Args[1:] { #C
        n, v, ok := strings.Cut(arg, "=")       #D
        if !ok {
            continue // can't parse the flag
        }
        parse, ok := parsers[strings.TrimPrefix(n, "-")]
        if !ok {
            continue // can't find a parser
        }
        if err = parse(v); err != nil {         #E
            err = fmt.Errorf("invalid value %q for flag %s: %w",
            break    // parsing error
        }
    }
    return err
}

func (f *flags) urlVar(p *string) parseFunc {   #F
    return func(s string) error {
        _, err := url.Parse(s)
        *p = s     #G
        return err
    }
}

func (f *flags) intVar(p *int) parseFunc {      #H
    return func(s string) (err error) {
        *p, err = strconv.Atoi(s)     #G
        return err
    }
}
```

Listing 5.5 first defines a new function type called `parseFunc` to keep the code short, understandable, and maintainable. Since each flag may have a different type, there are two type-specific parser providers: `urlVar` and `intVar`. They bind a parser to a variable and return a `parseFunc` parser that will update the variable via its pointer.

Tip

The `Errorf` function wraps an error and returns a new one. See this official blog post for more information: https://go.dev/blog/go1.13-errors.

The `parse` method puts the parsers in a `map` called the `parsers,` where the keys are the flag names and the values are the parsers. The loop parses each command-line flag and extracts it into the flag name and value pair. Then it uses the `map` and finds a registered parser by its name. Finally, it feeds the flag value to a parser responsible for parsing that flag.

Since you have a parser, let's try it as follows:

```
$ go run . -n=25 -c=5 -url=http://somewhere
...
Making 25 requests to http://somewhere with a concurrency level o
```

Cool!

Setting sensible defaults

However, the tool will print zero-values if you don't pass any flags:

```
$ go run .
...
Making 0 requests to  with a concurrency level of 0.
```

Providing sensible defaults to users makes it easy to use a command-line tool. For example, users can simply pass a url flag and omit the other flags. And let the tool decide how many requests to make and what concurrency level. This way, it will be convenient to use the tool instead of requiring users to pass all the flags all the time.

Let's give some sensible default values for the flags in Listing 5.6.

Listing 5.6: Integrating the parser (hit.go)

```
...
import (
    ...
    "runtime"
)

...

func main() {
```

```
    f := &flags{
        n: 100,                                 #A
        c: runtime.NumCPU(),          #B
    }
    ...
}
```

The `main` function in Listing 5.6 declares a new `flags` value with some sensible defaults:

- The number of requests is 100.
- And the default concurrency level depends on the number of CPUs on a machine.

Let's try it:

```
$ go run . -url=http://somewhere
...
Making 100 requests to http://somewhere with a concurrency level
```

Note

The concurrency level is 10 because my machine has 10 CPU cores.

You can overwrite the defaults using command line flags:

```
$ go run . -url=http://somewhere -n=25 -c=5
...
Making 25 requests to http://somewhere with a concurrency level o
```

Back to square one

Alright! Listing 5.5 in the previous sections was just an example of how to parse command-line arguments yourself. However, you'll use the flag package in the rest of the chapter.

Beware

You can now safely remove `parseFunc`, `urlVar`, and `intVar` from the `flags.go` file. Although you'll create the parse function from scratch using the flag package in the next section, please keep the parse function as it is.

5.3.3 The flag package

The hit is not a complex tool, so it can go a long way with the `os` package. However, you might want to make things easier down the road to keep command-line flag parsing more manageable. There are a few shortcomings, among others, when you use the `Args` variable:

- You need to parse arguments yourself.
- You need to do basic validation yourself.
- You need to provide the usage message yourself.

Enter the flag package: A package for parsing, validating, and displaying usage messages for command-line flags. It comes out of the box with support for parsing the string, int, float, time duration flags, etc. It is also extensible so that you can provide your own types.

This section will first teach you how the flag package works and how to define a flag using the flag package. Then, you'll define the flags for the hit tool using the flag package. Lastly, you'll refactor the flags to flag variables. Using flag variables will help you use the existing variables instead of the ones the flag package creates.

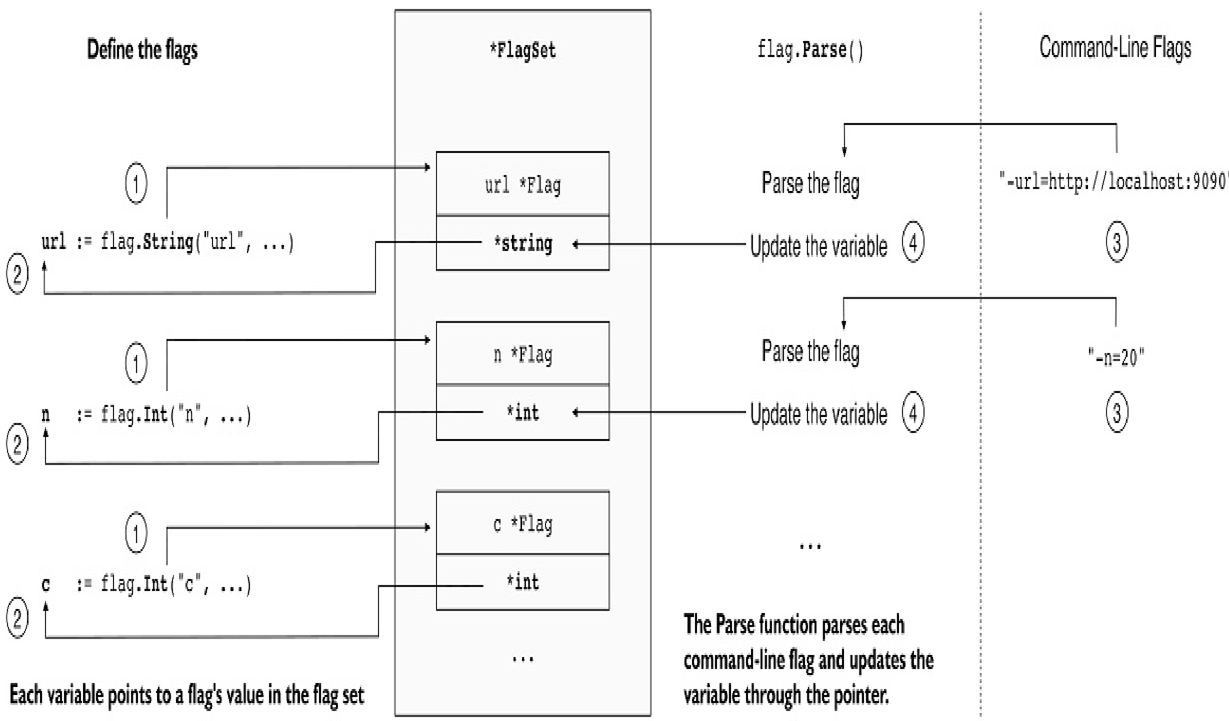
Parsing of flags

The flag package can parse the command-line flags for you. You first need to define your flags using one of the flag definition functions and then call the flag package's `Parse` function. Each flag definition function will return you a pointer so that you can see the final parsed values.

Note

A pointer stores the memory location of a value. For example, if you have a variable A, you can get its memory location and put that memory location into another variable called a pointer variable. The pointer variable will point to variable A's memory location. Since the pointer can find variable A through its memory location, you can also get variable A's value.

Figure 5.4 Define your flags and let the flag package parse them.



Note

The ellipses (...) in the flag definition functions denote the rest of the input values, such as a flag’s usage message and default value. I omitted them here for brevity and will explain them soon.

Let's discuss how it works in detail.

Step 1:

There are flag definition functions for some primitive types. For example, you can use the `String` function to parse the url flag to a string value, whereas the `Int` function can parse the number of requests flag (`-n` flag) to an integer value. For example, in Figure 5.4, the `String` function defines a string flag called `url`.

Note

There is a specific flag definition function for each type because you can get the parsed values as native Go values with a specific type, and the flag package does validation as well. For example, it won't accept a non-numerical value for an integer flag.

A flag definition function will create a flag definition for a command-line flag. And it will save the command-line flag's name, type, default value, and usage message in the flag definition (*not shown in the figure*). So, the flag package can know how it can parse the flag from the command line.

Then the flag definition function will save the flag definition in a structure called a `*FlagSet`. Think of a flag set like a bookkeeper: It keeps track of the flags you defined. There can be multiple flag sets, but the flag package uses a single one called `CommandLine`. For example, the flag package will define it on the `CommandLine` flag set when you define a flag.

Step 2:

A flag definition function will create an internal variable for a flag and return you a pointer to the internal variable. For example, in Figure 5.4, the `url` variable points to an internal `string` variable.

Think of a pointer as the flag package's way of sharing a flag's internal variable with you. If the flag package were to return a non-pointer, you wouldn’t see the flag's parsed value because it would only be updating the internal variable for a flag on its own.

Step 3:

Call the Parse function to let the flag package parse the command-line flags for you. It will extract each command-line flag you defined to name and value pairs. For example, in Figure 5.4, `"-url=http://localhost:9090"` becomes `"url"` (flag name) and `"http://localhost:9090"` (flag value).

Step 4:

The `Parse` function updates the internal variables of each flag, and you can see the extracted flag values since you have pointers to the internal variables.

Dereferencing a pointer will find the memory location of the internal variable and return its value. For example, in Figure 5.4, the `Parse` function will set the internal `string` variable's value to the parsed value (`"http://localhost:9090"`). So, you can get the extracted value via the `url` pointer variable by dereferencing it as follows: `*url`.

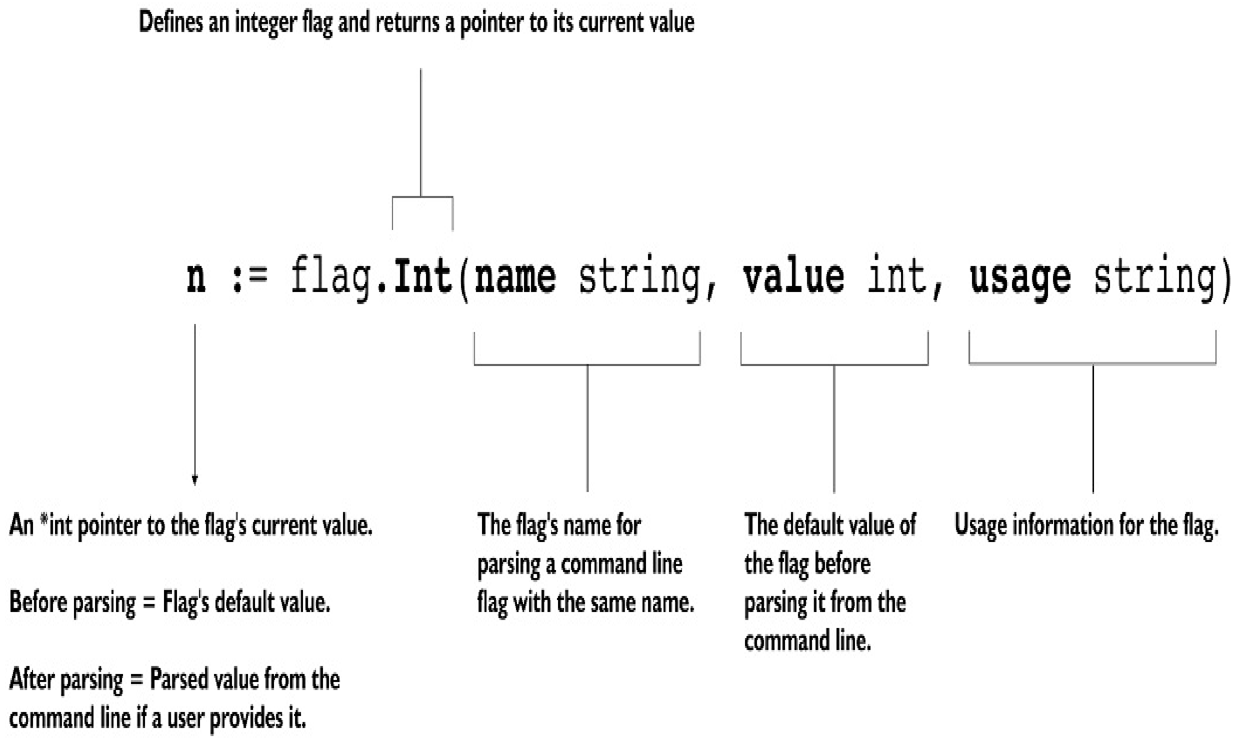
Tip

While declaring a pointer variable, you use: `var variableName *variableType` (e.g. `var url *string`). To get the value that is pointed by the pointer (dereferencing), you use: `*variableName`.

Defining a flag

Let's discuss what a flag definition function looks like in detail to understand better how it works and how you can define a new flag. Suppose you want to define the number of requests flag (an integer flag). So, you can use the `Int` function (Figure 5.5) to define a flag for parsing integers.

Figure 5.5 The Int function can define an integer flag.



The `Int` function makes an integer flag and returns an integer pointer to an *internal variable*:

- The `name` argument sets the flag's name to parse and print it in the usage message.
- The `value` argument sets the flag's default value to show in the usage message.
- The `usage` argument sets the flag's usage message for showing it to users.
- The `n` variable is a pointer to the internal variable with the flag's current value.

Since you learned how to define a flag using the `Int` function (Figure 5.5), it's time to define the flags for the hit tool using the flag package.

Parsing using the flag package

As shown earlier in the "Parsing of flags" section, there are several functions that you can use to define flags for various types. For example, you can use the `String` function to define a flag to parse a string and the `Int` function to define a flag to parse an integer.

You will define the following flags for the hit tool:

- `url`—A string flag without a default value.
- `n`—An integer flag with a default value of `20`.
- `c`—Another integer flag with a default value of `5`.

Listing 5.7 uses the `Int` and `String` functions to define all the flags you need.

Listing 5.7: Defining and parsing the requests flag (flags.go)

```
func (f *flags) parse() error {
   var (
       u = flag.String("url", "", "HTTP server URL to make reques
       n = flag.Int("n", 20, "Number of requests to make")
       c = flag.Int("c", 5, "Concurrency level")
   )
   flag.Parse()    #B
   f.url = *u      #C
   f.n = *n        #C
   f.c = *c        #C
   return nil
}
```

Listing 5.7 defines a string flag using the `String` function and the `Int` function for the integer flags. Each function returns a pointer to an internal variable that holds the default value you specified. Lastly, it updates the fields after calling the `Parse` function, so the `main` function can see them (e.g., `f.url = *u`).

For example, in Listing 5.7, `Int` defines a flag called `n` (at line 4), with a default value of `20` and a usage message of "`Number of requests to make`." The returned pointer `n`'s type would be `*int` and point to the flag's current value of `20`. The current value would be `20` because the `Int` function sets the flag's current value using the flag's default value (`20`). Then you could get the flag's current value by dereferencing the pointer (`*n` at line 9).

Let's run the program without providing any flags:

```
$ go run .
<<banner>>
Making 20 requests to  with a concurrency level of 5.
```

The `Int` function (at line 4) updated the flag's default value and you saw `20` when you executed the program. However, the `Parse` function did nothing because you didn't pass the `n` flag from the command line, yet. Let's do that next.

Note

The `Parse` function won't parse a flag if you don't pass the flag from the command line.

You will see the following when you pass `n` the flag:

```
$ go run . -n=1000
Making 1000 requests to  with a concurrency level of 5.
```

`Parse` parsed the `n` flag and set it to one thousand. And the concurrency level is five because five is its default value (see the line 5 in Listing 5.7).

You can also pass the `n` flag as follows:

```
$ go run . -n 1000
Making 1000 requests to  with a concurrency level of 5.
$ go run . --n=1000
Making 1000 requests to  with a concurrency level of 5.
```

All usages are the same, and the `Parse` function will set the `n` flag's value to `1000` in both cases. You can also provide the flags in a different order since *flags are not positional*:

```
$ go run . -n=5 -c=10
Making 5 requests to  with a concurrency level of 10.
$ go run . -c=10 -n=5
Making 5 requests to  with a concurrency level of 10.
```

The only difference is that the `Parse` function will parse the flags depending on the order you provide them from the command line. But it does not matter in this case because you'll get the same flag values no matter the order, with one exception: You can pass the same flag multiple times as follows:

```
$ go run . -c=2 -n=5 -c=4
...
Making 5 requests to  with a concurrency level of 4.
```

In the example above, the value of the `c` flag will be `4` instead of `2`. The `Parse` function sets the same flag twice in the order you passed the flags. It has first set the `c` flag to `2` and then set it to `4`. You will see the value behind this behavior when you learn about extending the flag package.

Note

The flag package will show a usage message when you pass the "`-h`" or "`-help`" flags from the command line or something goes wrong.

You can also see the usage as follows:

```
$ go run . –h
...
  -n int
        Number of requests to make (default 20)
```

As you can see, the usage message appears right below the flag and its default value is next to the usage message (`default 20`). Let's run it with all the flags:

```
$ go run . -url=http://localhost:9090 -n=2718 -c=30
Making 2718 requests to http://localhost:9090 with a concurrency
```

Setting the defaults

In Section 5.3.2 (Setting sensible defaults), you added some sensible defaults for the command-line flags, but Listing 5.7 did not use them. Let's bring them back in Listing 5.8.

Listing 5.8: Setting the provided defaults (flags.go)

```
func (f *flags) parse() error {
   var (
```

```
        ...
        n = flag.Int("n", f.n, "Number of requests to make")
        c = flag.Int("c", f.c, "Concurrency level")
    )
    ...
}
```

In Listing 5.8, there is no need to set the url field's default value as it does not use one. You only need to set the default values for the other fields.

Let's try it.

```
$ go run . -url=http://localhost:9090
Making 100 requests to http://localhost:9090 with a concurrency l
```

Let's print the usage message now:

```
$ go run . –h
Usage of .../hit:
  -c int
        Concurrency level (default 10)
  -n int
        Number of requests to make (default 100)
  -url string
       HTTP server URL to make requests (required)
```

Hooray! Cool, isn't it?

Changing a flag's type in a usage message

The usage message shows that the url flag expects a string value even though you're looking for a URL. Although every flag value is a string before parsing, you can tell people that you're expecting a URL string value by using a least-known trick. To do that, let's declare the url flag as follows:

```
u = flag.String("url", "", "HTTP server `URL` to make requests (r
```

When you wrap a flag usage message with back-quotes, the flag package will use that word to display it next to the flag name in the usage message as follows (The "`URL`" word in the flag's usage message stays the same as before (without back-quotes)):

```
$ go run . –h
...
  -url URL  <----
        HTTP server URL to make requests (required)
```

Updating the main function

The `Parse` function can print validation errors itself. Let's pass an invalid flag:

```
$ go run . -url=http://localhost:9090 -n=gopher
invalid value "gopher" for flag -n: parse error
<<usage message>>
```

Note

`Parse` ends the program if it fails or a user passes the `-h` or `-help` flags.

The `Parse` function fails, does not return to the `main` function, and terminates the program where you call it. Returning an error from the `parse` method in Listing 5.7 does not make sense at the moment. But, let's keep returning an error as you'll use it in the next sections.

Since the flag package handles the usage and error messages itself, let's remove the handling of the usage message and error printing from the main function in Listing 5.9.

Listing 5.9: Updating the main function (hit.go)

```
...
import (
    "fmt"
    "os"
    "runtime"
)

const bannerText = `...`     #A
// removes: usageText constant

func banner() string { ... } #A
// removes: usage()
```

```
func main() {
    f := &flags{
        n: 100,
        c: runtime.NumCPU(),
    }
    if err := f.parse(); err != nil {
        os.Exit(1) #B
    }
    fmt.Println(banner())    #A
    fmt.Printf("Making %d requests to %s with a concurrency level
        f.n, f.url, f.c)
}
```

Listing 5.9 keeps the banner code but removes the usage code. It also changes the previous `Fatal` function to the `Exit` function instead, as the `Parse` function can print the validation errors itself. The `Exit` function will be useful for post-parsing validation in the next sections.

Note

The `Exit` function crashes the program with a status code, but it won't print an error message, unlike the `Fatal` function.

Defining the flag variables

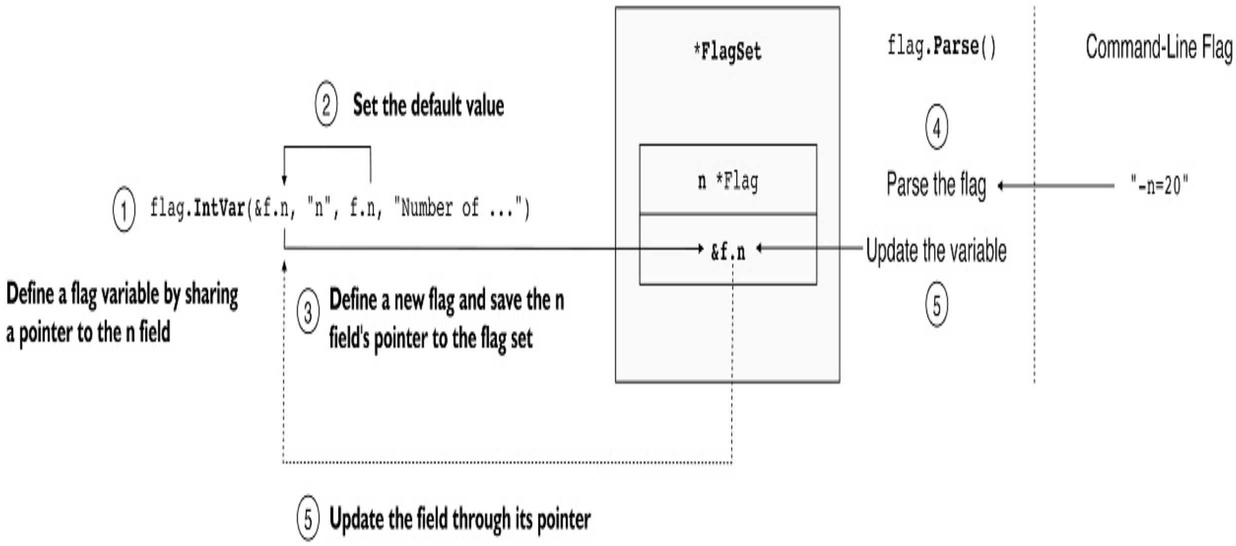
In the previous section, you defined the flags, and the flag package created a few internal variables and returned pointers to those variables. Then used the pointers to set the fields.

All systems are up and running, and the sky is clear. However, there is a problem. You have fields, and you can tell the flag package to use those fields instead of creating internal variables. Doing so will allow you to keep your code short and prevent possible mistakes of forgetting to assign to the fields.

You can use your variable if you pass its pointer as an additional first argument and add the `Var` suffix to a flag definition function.

Let me show you an example in Figure 5.6.

Figure 5.6 Passing an existing int pointer to the flag package.



1. Figure 5.6 passes the `n` field's pointer (`&f.n`) to the `IntVar` function.
2. `IntVar` sets the field to the default value. Here it sets the field onto itself! Not necessary here, but you could have given it a different default value.
3. `IntVar` defines the `n` flag, saves the pointer, and adds it to the default flag set.
4. `Parse` parses the command-line flag (`"-n=20"`) and extracts `20`.
5. Then it updates the `n` field via the pointer, and the field becomes `20`.

Note

The `IntVar` function connects a variable to a flag set. Unlike the `Var` function, the `IntVar` function does not return a pointer since you already passed one!

The flag package has a flag definition function for other flag types too. For example, there is a `StringVar` function for parsing string flags. And there is a `DurationVar` function for parsing time duration flags.

Let's update the `parse` method and pass the field pointers (Listing 5.10).

Listing 5.10: Using the flag variables (flags.go)

```
func (f *flags) parse() error {
   // You no longer need to declare variables

   flag.StringVar(&f.url, "url", "", "HTTP server `URL` to make r
   flag.IntVar(&f.n, "n", f.n, "Number of requests to make")
   flag.IntVar(&f.c, "c", f.c, "Concurrency level")
   flag.Parse()

   // Removes the field assignment code from this line.   #B

   return nil
}
```

In Listing 5.10, the `StringVar` function defines a string flag with a string pointer to the `url` field. And the `IntVar` function defines two integer flags for the remaining fields. You no longer need to set the fields yourself. The flag package will take care of them for you.

5.3.4 Wrap up

Let's talk a bit about what you have learned in this section:

- The os package allows you to parse raw command-line arguments using a slice variable called `Args`. When using the os package, you need to do the parsing, validation, and printing usage message yourself.
- The flag package offers a convenient way for handling command-line flags. You can use it to parse, validate, and print auto-generated usage messages for command-line flags.
- You can define flags by using functions like `Int`, `String`, etc. Each one returns a pointer variable that you can use to get the default and parsed values.
- Each defined flag will be in a structure called a `*FlagSet`. The default one is `flag.CommandLine`.
- You can pass a pointer to a flag definition function with a `Var` suffix like `IntVar` and `StringVar`. The flag package will update the variable through its pointer so that you don't need to update the variable yourself.
- A flag variable will have a default value you provided if a user doesn't provide it from the command line.
- The `Parse` function parses the command-line flags depending on the flags you define on a flag set. It terminates the program if an error

occurs or a user wants to see the user message.
- You can change a flag's type in the usage message by wrapping some parts of its usage message with backquotes.

5.4 Validating the arguments

You might want to fail fast when you detect a flag with an invalid value to make it easier for you and users to find the cause of a problem. It will also help you reduce bugs and make it convenient for users to use the tool.

Previously, you learned all the available options in the Go Standard Library for parsing the command-line arguments and flags. It's time to learn how to properly validate the command-line flags by combining the flag package and some home-grown methods.

First, you will learn how to validate the required flags. For example, the url is a required flag you cannot live without. You will add a custom validation function to make the flag a required flag.

After that, you will extend the validation function and validate for a flag that depends on another. Lastly, you will improve the parsing of the url flag and improve the validation error messages and make them more understandable to users.

5.4.1 Validating the required flags

The flag package can detect and handle invalid inputs such as type-mismatch and range errors. For example, the tool would fail with an error if you were to pass an invalid value to the number of requests flag:

```
$ go run . -n gopher
invalid value "gopher" for flag -n: parse error
$ go run . -n 99999999999999999999
invalid value "99999999999999999999" for flag -n: value out of ra
```

However, the flag cannot validate the missing flags. For example:

```
$ go run .
Making 100 requests to  with a concurrency level of 10.
```

```
----------------------^ !
```

As you can see, it didn't print an error and printed the url flag as an empty string instead.

Note

I managed to draw a naive ASCII graph that shows the location of the empty space.

Adding a validate method

The `Parse` function will update the `url` field when it finishes parsing. You can take advantage of this fact and create a method that returns an error if the `url` field is empty (Listing 5.11).

Listing 5.11: Making the url field mandatory (flags.go)

```
...
import (
    "errors"
    "flag"
    "fmt"
    "os"
    "strings"
)
...
func (f *flags) parse() error {
    ...
   flag.Parse()
   if err := f.validate(); err != nil {         #A
     fmt.Fprintln(os.Stderr, err)     #B
     flag.Usage()             #C
     return err    #D
   }
   return nil
}

// validate post-conditions after parsing the flags.
func (f *flags) validate() error {
    if strings.TrimSpace(f.url) == "" {         #A
        return errors.New("-url: required")     #A
    }   #A
```

```
    return nil
}
```

Let's try it:

```
$ go run .
-url: required
<<usage>>
```

Listing 5.11 declares a new `validate` method instead of handling the validation logic in the `parse` method so that each method is responsible for one thing. It's also easier to read and manage it this way.

The `parse` method can now return an error if the `url` field is still empty after parsing the command-line flags. This situation can only happen if a user does not provide the url flag. When an error occurs, the `parse` method will first print the error to the standard error (`os.Stderr`) stream using the `Fprintf` function and then print the usage message.

Note

The flag package writes the error and usage message to the standard error stream. The `validate` function follows in its footsteps and does the same.

`Fprintln` is like `Println`. The only difference is that it takes a value that satisfies the `io.Writer` interface. You can pass any value with a `Write` method to the `Fprintf` function as the first argument because the `Writer` interface has a single method: `Write(...)`. The standard error has a `Write` method so that you can pass it to the function.

The `Usage` function forces the flag package to show the usage message. You need to call it yourself because the flag package thinks there was no error in parsing.

The flag package would terminate the program if there was an error in parsing or a user wanted to see the usage message. However, it wouldn't do so here since there wasn't an error. So the program would go on as if nothing had happened. Returning an error from the `validate` method prevents the problem so the `main` function can terminate the program.

The `TrimSpace` function removes spaces surrounding a string. Users could pass the url flag with multiple spaces, and the validate method wouldn't detect the problem if Listing 5.11 didn't use the `TrimSpace` function. You would see the following if it wasn't validating the flag:

```
$ go run . -url="    "
...Makes 100 requests to       with a concurrency level of 10...
```

Fortunately, the `TrimSpace` function prevents the problem by removing spaces. You now have a robust flag validator. Well done!

Validating the concurrency flag

The concurrency flag should be less than or equal to the number of requests flag. Why? Imagine what would happen if a user wanted to make one request with two goroutines. Does it make sense? Only one goroutine can execute code and make a single request. What would the other one do? Eat, forage, sleep underground as all lovely gophers do?

To do that, you can add another check to the `validate` method as in Listing 5.12.

Listing 5.12: Validating the concurrency flag (flags.go)

```
func (f *flags) validate() error {
    if f.c > f.n { #A
        return fmt.Errorf("-c=%d: should be less than or equal to
    }   #A
    ...
    return nil
}
```

The `validate` method now checks and returns an error if the concurrency level exceeds the number of requests. It also saves the parsed values of the flags in the error so that a user can see what was wrong with them:

```
$ go run . -url=http://somewhere -n=10 -c=20
-c=20: should be less than or equal to -n=10
<<usage>>
```

It looks good to me!

5.4.2 Validating the url flag

You can use the `validate` method for validating the url flag. However, instead of doing that in the `validate` method, let's do it in a new function that you can use to parse the url flag and return an error if parsing fails (Listing 5.13). This way, you can keep the `validate` method cleaner.

Listing 5.13: Validating the url flag in another method (flags.go)

```
// validate post-conditions after parsing the flags.
func (f *flags) validate() error {
    ...
    // Removes the url validation from here (moves it to validate

    if err := validateURL(f.url); err != nil {
        return fmt.Errorf("invalid value %q for flag -url: %w", f
    }
    return nil
}

func validateURL(s string) error {
    if strings.TrimSpace(s) == "" {
        return errors.New("required") #B
    }
    _, err := url.Parse(s)   #C
    return err
}
```

The `validate` method in Listing 5.13 parses the `url` field using the `validateURL` function. The `validate` method also improved the error message for a consistent user experience of the hit tool (*makes the error similar to the error messages of the flag package*):

```
$ go run .
invalid value "" for flag -url: required
$ go run . -url=://
invalid value "://" for flag -url: parse "://": missing protocol
```

Improving the error messages

The previous `"missing protocol scheme"` error message is verbose. Let's trim it down in Listing 5.14 because the flag package already reports a similar error. It will also simplify the other possible errors with its own versions.

Listing 5.14: Improving the error messages (flags.go)

```
func validateURL(s string) error {
    u, err := url.Parse(s)   #A
    switch {
    case strings.TrimSpace(s) == "":
        err = errors.New("required")
    case err != nil:         #B
        err = errors.New("parse error")         #B
    case u.Scheme != "http": #C
        err = errors.New("only supported scheme is http") #C
    case u.Host == "":       #D
        err = errors.New("missing host")        #D
    }
    return err     #E
}
```

The `validateURL` function in Listing 5.14 checks whether there was an error or not, but it also checks for some other unwanted and probable situations because of invalid input. For example, the hit tool will only support the HTTP protocol. It will return an error if a user wants to use another protocol. Let's take a look at the error messages:

```
$ go run . -url=://
invalid value "://" for flag -url: parse error
$ go run . -url=ftp://
invalid value "ftp://" for flag -url: only supported scheme is ht
$ go run . -url=http://
invalid value "http://" for flag -url: missing host
```

Better!

5.4.3 Wrap up

- You need to do post-parsing validations yourself as the flag package cannot check for the missing flags or other post-parsing conditions.
- The `Usage` function forces the flag package to show the usage message. You can use it to print the usage message if post-parsing validation fails.

- The flag package writes the errors and usage message to the standard error.

5.5 Extending the flag package

You learned how to define your flags using the flag package. Then you defined the flags for the hit tool. You also learned to validate the flags using the flag package and a custom validation method you wrote.

The number of requests and the concurrency level flags should not be zero or negative. They need to be natural numbers (greater than zero). You could add more conditions to the custom validation function you wrote earlier to check if those flags were positive. However, it would slightly complicate the validation function, and it can become unmaintainable in the future if you were to add more similar flags.

The Go Standard Library's flag package is versatile, so you can easily extend it with custom flag types. This section will teach you how to extend the flag package with a custom flag type that will only accept natural numbers. Doing so will leave some part of the validation to the flag package instead of manually doing it all yourself.

Once you learn to extend the flag package, you can add more flag types. For example, you could add a new flag type to parse a comma-separated value to a slice. Say you want to use the hit tool to make requests to multiple URLs (which you won't do in this book):

```
$ go run . -urls=http://a.com,http://b.com,http://c.com
```

You could add a new flag type to let the flag package parse the urls field into a slice:

```
[]string{"http://a.com", "http://b.com", "http://c.com"}
```

You could also validate each URL in your new flag type. Or, you could create a new flag type to parse a command-line flag to a `*url.URL` value.

5.5.1 Inner mechanics

Before extending the flag package, you need to learn how it works. Let's get started by learning about the inner mechanics of the flag package. As you know, a command-line flag consists of two parts:

1. The flag name.
2. The flag value.

Both are `string` values.

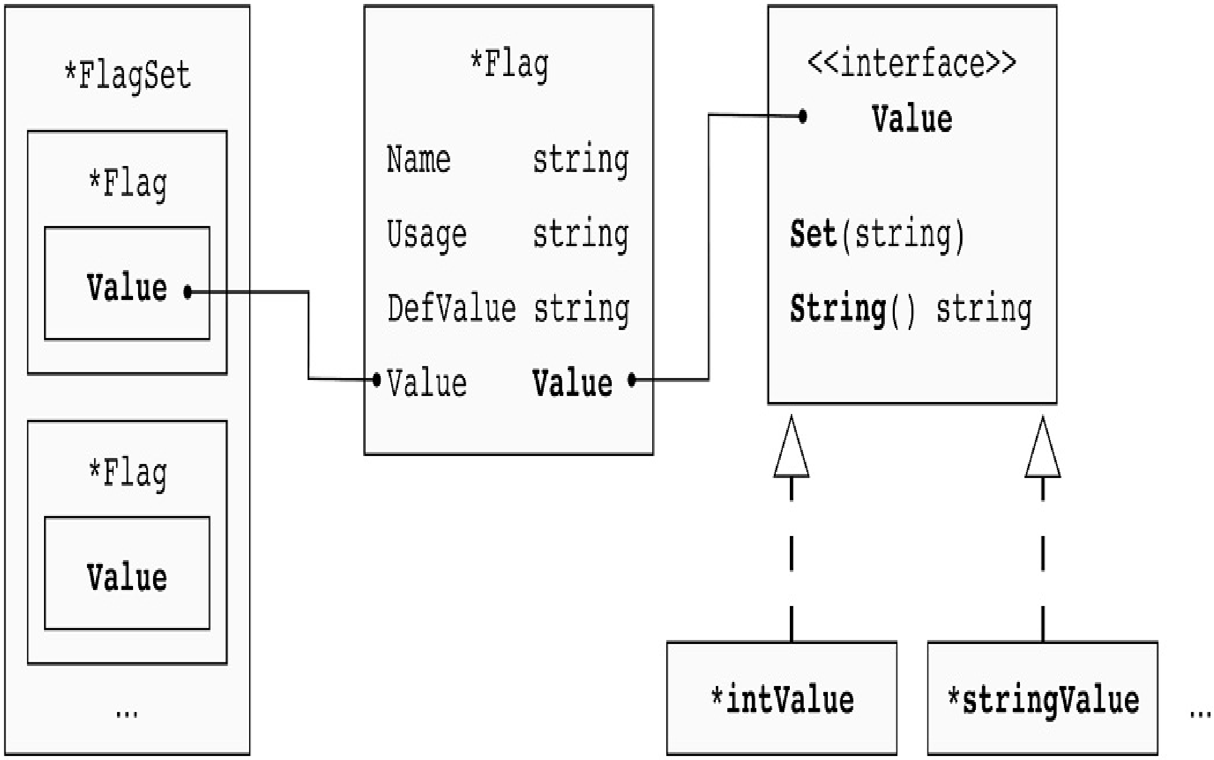
However, the `Parse` function can set different types of variables. For example, it can get `"-n=20"` from the command line, and then it can set the `n` integer field to `20`. Or, it can get `"-timeout=1s"` and then set it to a `Duration` variable. It can also work with other types.

Have you ever wondered how? Here it comes!

The Value interface

The flag package can work with various types thanks to the `Value` interface. As shown in Figure 5.7, each flag on a flag set is a `*Flag` value that contains details such as the flag's name, usage message, and default value. Each one also has a dynamic value that satisfies the `Value` interface.

Figure 5.7 Each flag has a dynamic value type that satisfies the Value interface.



Implementations of the Value interface

Note

I'll say a "variable" when referring to a variable you use to define a flag. For example, the `n` field is a variable. And I'll say a "pointer" when referring to the variable's pointer. For example, the `n` field's pointer is `&f.n`.

The `Value` is a simple interface, and it has only two methods:

- The `Set` method takes a command-line flag value, converts it, and sets a variable through its pointer. The `Parse` function uses this method for setting a flag's variable.
- The `String` method gets the variable through its pointer, converts it, and returns it as a `string` value. The `Var` function (you'll see in a bit!) uses this method to set the flag's default value.

The `Set` and `Parse` functions work with string values because command-line flags are strings. Underlyingly, they convert them to other types.

Tip

An idiomatic interface has one or few methods.

The flag package hands over the responsibility of parsing and getting the command-line flags to types that satisfy the `Value` interface. For example, the `stringValue`, `intValue`, `float64Value`, `durationValue`, etc., all are implementations of the `Value` interface. Each one satisfies the `Value` interface and has a `String` and `Set` method.

Tip

Ideally, an interface should only contain behavioral methods. For example, each flag in a `FlagSet` is a concrete `Flag` value instead of a `Value` interface value. Only a `Flag` value's `Value` field is an interface value because only the `Value` field should be abstract and dynamic. The other fields, such as `Name`, `Usage,` and `DefValue,` should be stored as concrete values because they are not about behavior but a state.

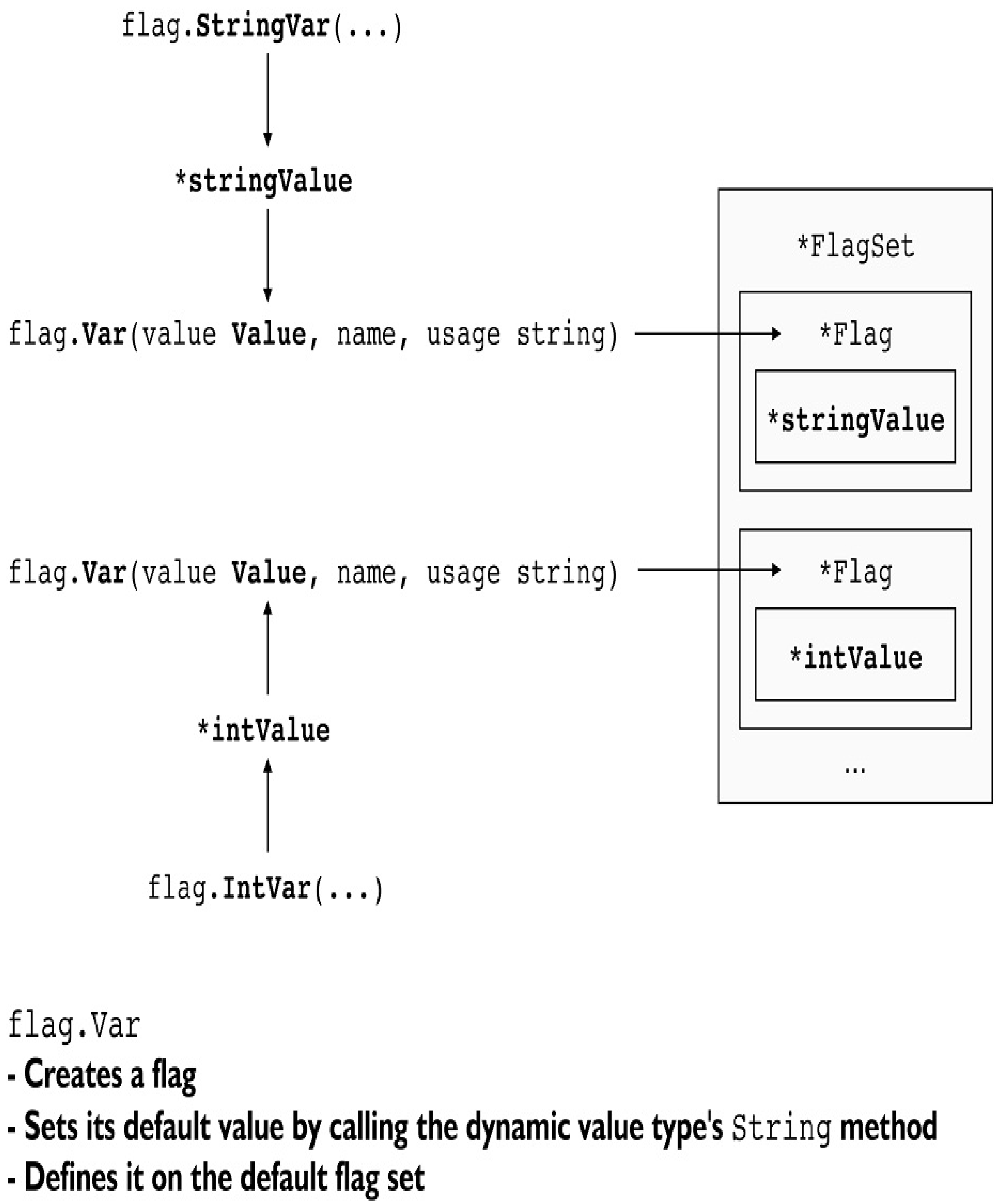
The Var function

How can the flag package define a flag with a dynamic value type, such as a `stringValue` or `intValue`? It's time to learn how it works.

Note

Behind the scenes, all the flag definition functions use the `Var` function.

Figure 5.8 The Var function can create a new flag with a dynamic value type on the default flag set.



The `Var` function can define a flag on the default flag set with its dynamic value. It takes a `Value` interface value, a flag's name, and its usage string.

For example, in Figure 5.8:

- The `StringVar` function creates a `*stringValue`.
- Then, it passes the `*stringValue` and other flag details to the `Var` function.
- Finally, the `Var` function creates a new flag and sets the flag's default value by calling the `String` method on the `*stringValue`.
- Then it defines the flag with its default value on the default flag set.

Note

Unlike the other flag definition functions, the `Var` function does not take a default value because a dynamic value type is responsible for setting it.

Wrap up

This section taught you how you can extend the flag package.

Let's see what you have learned:

- Underlyingly, all flag definition functions use the `Var` function.
- The `Var` function can define a flag on the default flag set with a dynamic value.
- A dynamic value satisfies the `Value` interface.
- The `Value` interface has a `Set` and `String` method.
- The `Set` method gets a string value, converts it, and sets a variable through its pointer. The `Parse` function uses the `Set` method.
- The `String` method gets a value, converts it to a string, and returns it. The `Var` function uses the `String` method to set a flag's default value.

5.5.2 Creating a flag type

Like I said in the section introduction, you'll declare a flag type that only accepts natural numbers for the number of requests and the concurrency level flags.

To do that:

1. You need to create a new type that satisfies the `Value` interface.
2. Then you need to register it to the default flag set using the `Var` function.

By doing so, the `Parse` function can do its magic.

What does the number type look like?

Since you want to create a type that accepts only the natural numbers, why don't we call it simply a `number`?

Note

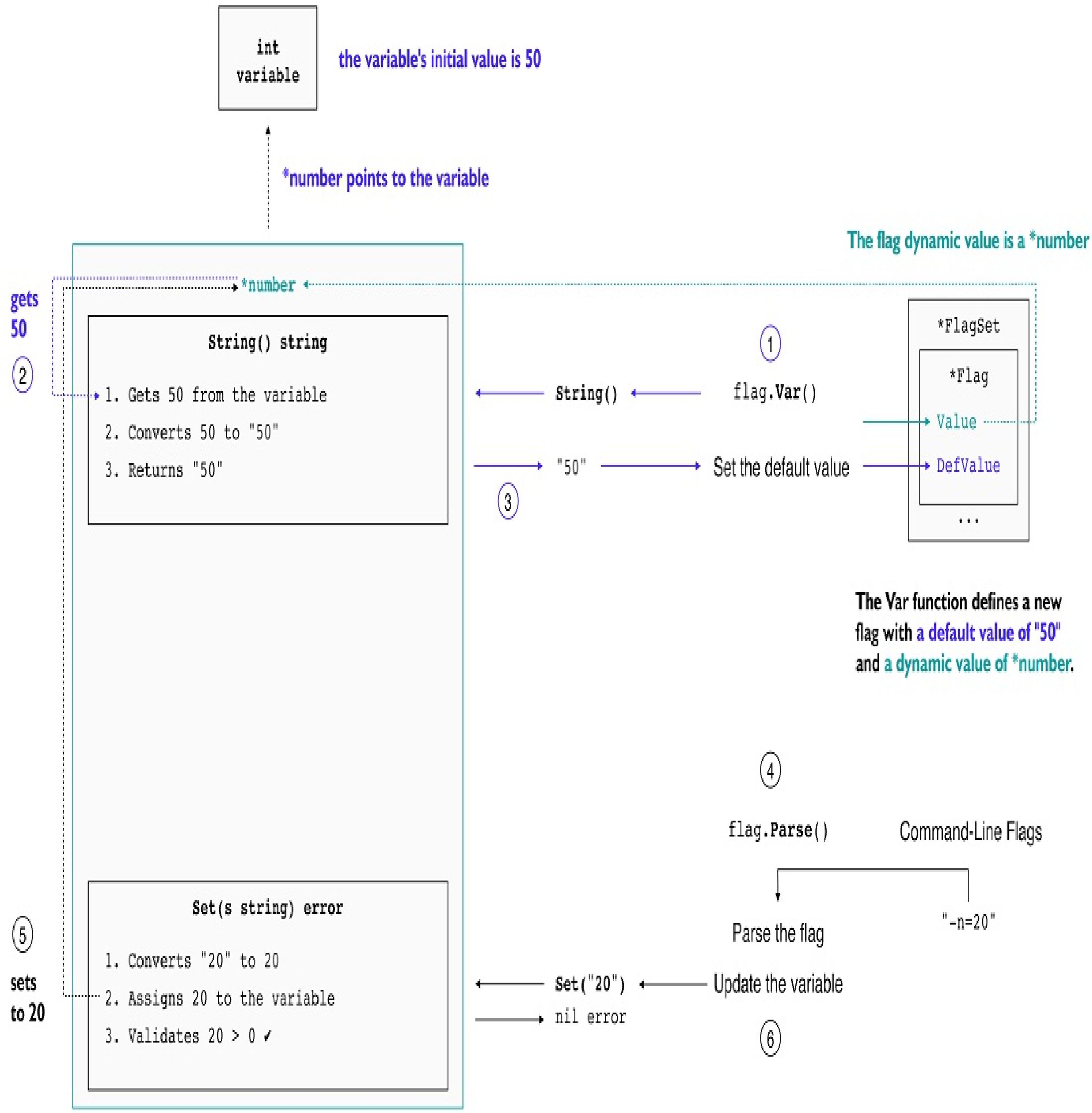
A struct field is also a variable.

Say you define a new flag using an `int` variable, and the variable's value is `50`, as follows:

```
// f.n field is an int variable and equals 50
// toNumber(&f.n) returns a number pointer to f.n's memory addres
flag.Var(toNumber(&f.n), "n", "Number of requests to make")
```

The `toNumber` function returns a `*number` that points to the variable's memory location.

Figure 5.9 The creation of a new flag and how it can read from and set a variable via a dynamic value type.



In Figure 5.9, `*number` points to the `int` variable, and the variable's current value is `50`.

1. `Var` makes a new flag and calls the `number`'s `String` method.
2. `String` reads the variable via its pointer and converts its value to `"50"`.
3. `Var` gets `"50"` from the `String` method, sets the flag's default value to `"50"` and dynamic value to the `*number` value. Finally, it defines the flag on the default flag set.

Say you call the `Parse` function.

1. It parses `"-n=20"` and extracts `"20"`. Then it calls `*number`'s `Set` method to update the variable.
2. `Set` converts `"20"` to `20` and assigns it to the variable via its pointer.
3. Then it returns a `nil` error because `20` is positive. If the number was zero or negative, `Set` would return an error, and `Parse` would print the error and the usage message.

You now know how the flag package uses a dynamic value type. Let's make your own type!

Listing 5.15: Adding the number value type (flags.go)

```
// number is a natural number.
type number int    #A

// toNumber is a convenience function for converting p to *number
func toNumber(p *int) *number    {
    return (*number)(p)      #B
}

func (n *number) Set(s string) error {
    v, err := strconv.ParseInt(s, 0, strconv.IntSize)     #C
    switch {
    case err != nil:
        err = errors.New("parse error")
    case v <= 0:
        err = errors.New("should be positive")
    }
    *n = number(v) #D
    return err
}

func (n *number) String() string {
    return strconv.Itoa(int(*n))      #E
}
```

Listing 5.15 makes a new dynamic value type called `number`.

- The `*number` type has `Set` and `String` methods and satisfies the `Value` interface.
- The `toNumber` function returns a `*number` from an `int` variable's pointer.

It can read from and write to the `int` variable via the pointer.
- `Set` sets the variable and returns an error if the flag value is zero or negative.
- `String` returns the variable's current value as a `string`.

strconv.ParseInt vs. strconv.Atoi

Both `ParseInt` and `Atoi` functions can convert a `string` to an `integer`. However, the `ParseInt` function is superior. For example, it can parse `1_000` to `1000` or `0xff` to `255`. See the link: https://pkg.go.dev/strconv#ParseInt.

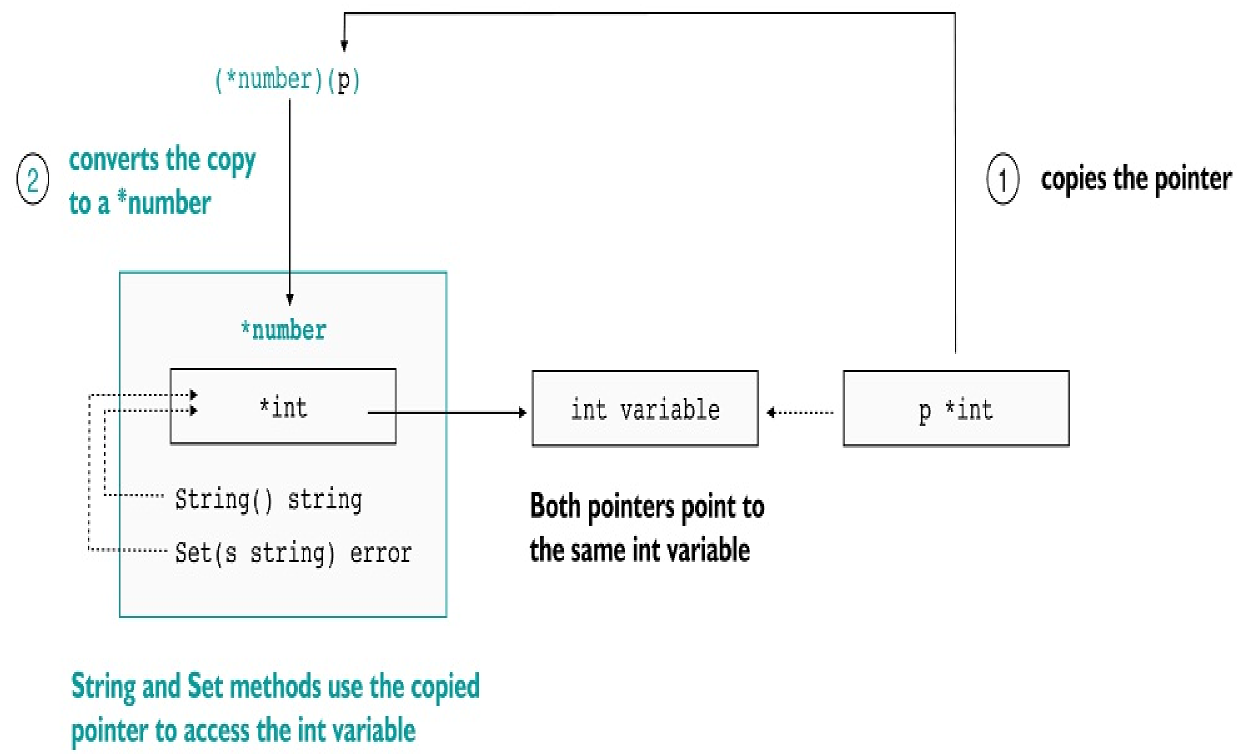
Adding methods on the fly

Let's pause a little bit here as something interesting happens in Listing 5.15. It converts the input value `p` to a `*number` value in the `toNumber` function as follows:

```
return (*number)(p)
```

Doing so puts the `*number`'s methods on the `int` pointer. How does it work?

Figure 5.10 Adding methods on the fly. Both pointers point to the same variable.



1. It copies the pointer. Both `p` and copy point to the same memory address where the program stores the `int` variable.
2. The copy obtains the `String` and `Set` methods since it becomes a `*number` pointer. The `Set` and `String` methods can write to and read from the `int` variable via the `*number` pointer.

Note

The `*number` and `*int` values point to the same variable's memory address.

For example, you can create a new `number` pointer using an `int` pointer as follows:

```
var i int      // 0
n := toNumber(&i)
```

Then you can set the original variable to another value using the `Set` method:

```
n.Set("20")
fmt.Println(i) // 20
```

The `Set` method can do so because it knows the memory address of the original variable. You can also print the variable using the `String` method as follows:

```
fmt.Println(n.String()) // 20
```

You don't have to call the `String` method to print it:

```
fmt.Println(n)          // 20
```

It works because a dynamic value type is also a `Stringer`:

```
type Stringer interface {
    String() string
}
```

The types that satisfy the `Value` interface also satisfy the `Stringer` interfaces. `Println` will call the `String` method whenever it detects an input value is a `Stringer`.

Tip

Decent Go interfaces are small and composable from other interfaces.

Defining flags using a custom type

It's time to update the `parse` method and use the new dynamic type (Listing 5.16).

Listing 5.16: Defining the flags with a custom type (flags.go)

```
func (f *flags) parse() error {
    ...
    flag.Var(toNumber(&f.n), "n", "Number of requests to make")
    flag.Var(toNumber(&f.c), "c", "Concurrency level")    #A
    flag.Parse()
    ...
}
```

Remember

You need to use the `Var` function to define a flag with a dynamic type.

Let's try it!

```
$ go run . -url=http://localhost -n=0
invalid value "0" for flag -n: should be positive
$ go run . -url=http://localhost -c=0
invalid value "0" for flag -c: should be positive
$ go run . -url=http://localhost -n=1_000_000 -c=0xff
Making 1000000 requests to http://localhost with a concurrency le
$ go run . -url=http://localhost -n=gopher
invalid value "gopher" for flag -n: parse error
```

Let's try it with the default values:

```
$ go run . -url=http://localhost
Making 100 requests to http://localhost with a concurrency level
```

Remember

The default values come from the main function.

5.5.3 Wrap up

You can now define your own dynamic value types.

- The flag package is versatile and extendable.
- Dynamic value types satisfy the `Value` interface.
- A dynamic value type lets you customize the parsing and validation of command-line flags.
- The `Var` function can define a flag with a dynamic value on the default flag set.
- In Go, you can add methods to a value on the fly.

5.6 Using a positional argument

Users can use flags as *optional parameters* to change the behavior of the hit tool. However, the url flag is not one of them since the tool always needs one for making HTTP requests. It might be a good idea to make it a positional argument instead of an optional flag. This way, you'll convey users the idea that the url is always required and it's not an option.

You will first understand the differences between flags and positional arguments. Then, you will customize printing of the usage message because the flag package cannot know anything about an argument if you don't define it as a flag.

After that, you'll get the url argument from the command-line yourself. The flag package will also help you doing that because it supports getting command-line arguments too.

Flags vs. positional arguments

Let's get started by understanding the differences between flags and positional arguments.

Currently, the url flag can be in any position:

```
$ go run . -url=http://foo.com -c=1 -n=10
$ go run . -c=1 -n=10 -url=http://foo.com
```

Remember

The flag package parses the flags in the order you pass from the command line.

A *positional argument*, on the other hand, is sensitive to its position—hence the name:

```
$ go run . http://foo.com
$ go run . -c=1 -n=10 http://foo.com
$ # the following is not allowed:
$ go run . http://foo.com -c=1 -n=10
```

Note

The flag package will parse the positional argument only after parsing the flags.

As you can see, the url argument should be the last argument you will be passing. Otherwise, the flag package won't parse the flags because the parsing stops just before the first non-flag argument. The url argument's position should be fixed and it should be the last argument.

Note

You can read more about this style of handling command-line arguments (the POSIX standard) in the link: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html

Making the url a positional argument

Since you understand the differences, let's make the url flag a positional argument. You will customize the usage message by using a function called `Usage` from the flag package. The flag package sets the `Usage` function by default.

Fortunately, you can first add your customized message and then tell the flag package to print the usage message for the flags too. To do that, you'll use another function called `PrintDefaults`. The `PrintDefaults` function tells the flag package to print the usage message of the flags you defined.

After customizing the usage message, you'll remove the url flag definition since you'll be handling the url argument yourself. Lastly, you'll use a function called `Arg` from the flag package to get the first argument after the flags.

Since you know a little bit about the new functions you'll be using, if you're ready, let's make the url flag a positional argument in Listing 5.17.

Listing 5.17: Switching to a positional argument (flags.go)

```
...
const usageText = `#A
Usage:
  hit [options] url
Options:`
...
```

```
func (f *flags) parse() error {
    flag.Usage = func() {    #B
        fmt.Fprintln(os.Stderr, usageText[1:])  #C
        flag.PrintDefaults() #D
    }

    // Removes the url flag by removing the StringVar call.

    flag.Var(toNumber(&f.n), "n", "Number of requests to make")
    flag.Var(toNumber(&f.c), "c", "Concurrency level")
    flag.Parse()

    f.url = flag.Arg(0)      #F

    if err := f.validate(); err != nil {
        ...
    }
    ...
}
```

The flag package is about flags, but not the command-line arguments, and it cannot tell users how to print the usage message of a positional argument. You need to do that yourself.

In Listing 5.17:

1. The `parse` method customizes the usage message by calling the `Usage` function.
2. It prints its own usage text to the standard error (where the flag package prints).
3. Then it tells the flag package to print the usage message of the flags by calling the `PrintDefaults` function.

You can get a command-line argument using the `Arg` function, and `Arg(0)` is the first remaining argument after parsing the command-line flags.

1. Listing 5.17 removes the url flag by removing the `StringVar` function.
2. It passes zero to the `Arg` function and gets the url argument.
3. Finally, it sets the `url` field since the flag package no longer handles the url flag.

Tip

The `Arg` function returns a command-line argument after parsing flags. You don't need to check the length of the arguments (as you would do when using the `os.Args` variable) since it returns an empty string if the argument at an index is missing.

The final usage message looks like the following:

```
$ go run . –h
Usage:
  hit [options] url
Options:
  -c value
        Concurrency level (default 10)
  -n value
        Number of requests to make (default 100)
```

Looks fine! Before finishing the section, let's improve the error message in the `validate` method.

Listing 5.18: Improving the error message (flags.go)

```
func (f *flags) validate() error {
    ...
    if err := validateURL(f.url); err != nil {
        // instead of: invalid value %q for flag -url: %w
        return fmt.Errorf("url: %w", err)
    }
    ...
}
```

Since the url is an argument and not a flag, Listing 5.18 simplifies its error message. Previously, it was printing an error message about the url flag:

```
invalid value %q for flag -url: %w
```

Finally, let's try the hit tool as follows:

```
$ go run .
url: required
$ go run . http://
url: missing host
$ go run . http://localhost:9090
Making 100 requests to http://localhost:9090 with a concurrency l
```

```
$ go run . -n=50 -c 5 http://localhost:9090
Making 50 requests to http://localhost:9090 with a concurrency le
```

That's it! The hit tool is easier to use than ever before. What do you say?

5.7 Testing the CLI tool

Testing a CLI tool should not be different from testing any other code. Then again, people in the wild use many weird techniques to test their CLI tools and complicate things. Read on if you don't want to join them.

In this section, you will learn the following:

- Making the main function testable.
- Testing the CLI tool.
- Parallel testing.
- Using a custom flag set.

Let's get started!

5.7.1 Testing the main function

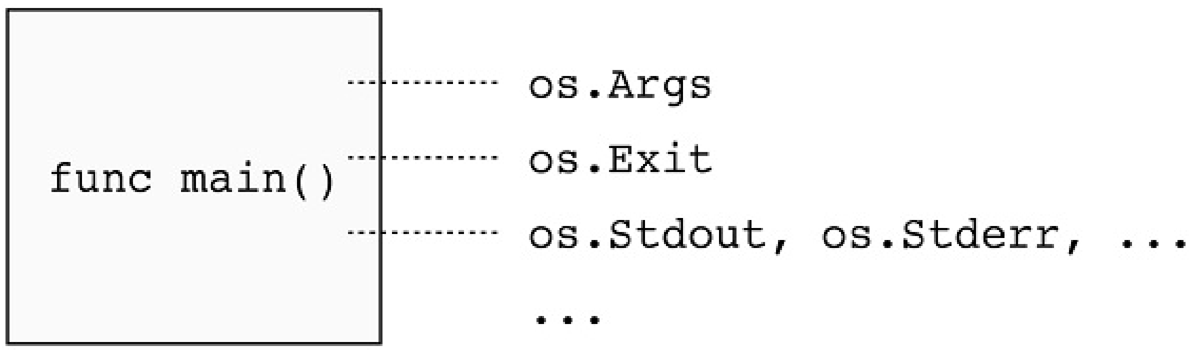
Why is the main function not testable? Or is it? There are some hacky ways for testing it:

- Compiling and executing the program using the stdlib's exec package.
- Setting the `os` package's `Args` variable to a temporary variable and then switching it back to the original.
- Changing and tracking the standard out and standard error with something else.

The list goes on. These hacky ways can sometimes be necessary, but it depends on your imagination. In this section, you'll learn how to make the main function testable instead of using hacky ways.

Why is the main function not testable?

Figure 5.11 The main function is not easily testable because it uses globals.



So let's get back to square one and ask again: Why is the main function not testable? As shown in Figure 5.11, the main function uses globals, and globals are often out of your control.

For example:

- The `Println` function and friends use the standard output stream.
- The `Fatal` function and friends use the standard error stream.
- The `Exit` function uses the standard error stream.
- The flag package uses the `Args` variable and the standard error stream.

Note

The `main` function itself does not dictate using global values.

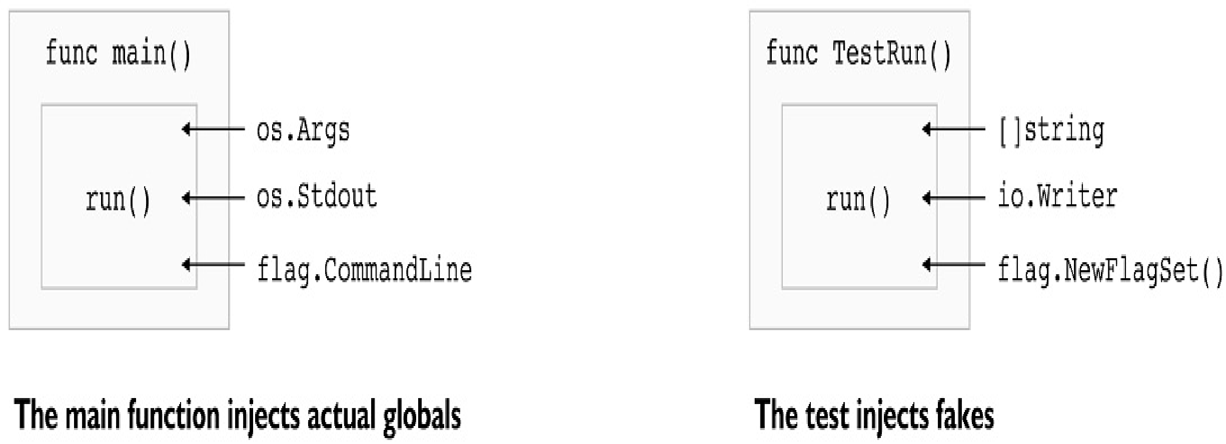
To be honest, it's not the main function's fault but the programmers because they use globals within it (or worse in other places).

You need to take control of these values to effectively test the CLI tool. Doing so would be especially useful for parallel testing and running tests faster. For example, the default flag set is a singleton for each program, which is not good for parallel testing. You cannot parallel test when you use a global value. Otherwise, you'll have to embrace concurrency issues and deal with mutexes and channels.

How can you make it testable?

You can make the main function testable by extracting it and putting it into another function. Then you can call the new one from the main function. Nothing would change. However, it would change the entire scenery if you could inject the command-line arguments, standard streams (standard error and standard out), and others into the new one.

Figure 5.12 The run function gets the environmental values from outside. It is easily testable because it does not interact with any global value.



You can see a new function called `run` in Figure 5.12. It gets environmental values from outside, such as command-line arguments, output streams, and even a flag set. This approach makes it effectively testable because you can run it in a controlled environment.

You can use the `run` function from the main function and feed it with the real environment values. And you can provide it with fake values while testing it. So the `run` function can both run in a real and an isolated test environment. Kill two drones with one stone, again.

5.7.2 Creating the run function

In Listing 5.19, the `main` function injects the globals to the `run` function, and the `run` function is responsible for parsing the flags, printing the banner, integrating with the hit package, etc.

Listing 5.19: The run function (hit.go)

```
...
import (
    "flag"
    "io"
    ...
)

func main() {      #A
    if err := run(flag.CommandLine, os.Args[1:], os.Stdout); err
        os.Exit(1) #C
    }
}

func run(s *flag.FlagSet, args []string, out io.Writer) error {
    f := &flags{
        n: 100,
        c: runtime.NumCPU(),
    }
    if err := f.parse(s, args); err != nil {    #E
        return err #F
    }
    fmt.Fprintln(out, banner())       #G
    fmt.Fprintf(out, "Making %d requests to %s with a concurrency
        f.n, f.url, f.c)

    // hit pkg integration here

    return nil
}
```

The `main` function in Listing 5.19 runs the `run` function and returns a status code of one to the operating system if the `run` function fails.

The `run` function is now a flexible replica of the earlier `main` function:

- Instead of the global one, it gets a flag set to work with any flag set. The `main` function passes it the global flag set called `CommandLine`. However, you'll pass a new flag set value to the `run` function while testing it.
- It gets a custom set of command-line arguments. You'll pass fake arguments while testing it.
- It gets a custom writer. The `main` function passes the standard out, while a test can pass a fake writer.

Tip

You don't have to test the main function if you keep it short.

The `main` function becomes extremely simple, and there is no need to test it. Simple manual testing can validate whether it works. What you should be testing is the `run` function. You can go crazy and test it against every condition out there.

Updating the parse method

The `run` function passes a custom flag set to the `parse` method in Listing 5.19. You will update the `parse` method next and make it accept a flag set and command-line arguments in Listing 5.20.

Listing 5.20: Fixing the parse method (flags.go)

```
func (f *flags) parse(s *flag.FlagSet, args []string) error {
    s.Usage = func() {
        fmt.Fprintln(s.Output(), usageText[1:]) #B
        s.PrintDefaults()    #C
    }

    s.Var(toNumber(&f.n), "n", "Number of requests to make")
    s.Var(toNumber(&f.c), "c", "Concurrency level")       #C

    if err := s.Parse(args); err != nil {       #D
        return err
    }
    f.url = s.Arg(0)                            #E

    if err := f.validate(s); err != nil {       #F
        fmt.Fprintln(s.Output(), err) #B
        s.Usage()  #B
        return err
    }

    return nil
}
```

The `parse` method in Listing 5.20 now uses the given flag set instead of the default one.

Note

The `main` function would still pass the default flag set to the `run` function when running the hit tool (Listing 5.19). However, you can now pass a custom flag set while testing it. Yay!

The `parse` method now parses the command-line flags and passes the arguments to the given flag set's `Parse` method instead of the flag package's global `Parse` function.

Note

The flag package's global `Parse` function does not take a command-line arguments input value because it uses the os package's `Args` variable by default.

The `Fprintln` function prints to the given flag set's writer (`s.Output()`) instead of directly writing to the standard error so that you can control where it writes while you're testing it.

The `Usage` *function* was writing to the standard error. Since the `main` function passes the default flag set to the `run` function, the `Usage` *method* would also write to the standard error. Fortunately, you can change this behavior while testing it because the `Usage` method prints to the flag set's writer instead.

5.7.3 Testing the run function

Finally, it is time to test the `run` function. Since the function gets everything from outside, you can feed it with fake values and test it against them. For example, you can feed it an in-memory bytes buffer and check whether it contains what you're looking for. You can also feed it with fake command-line flags to see how it behaves.

You'll first write tests to test the run function's happy path. You'll test whether it gives you an error when you use it correctly. Then you'll write tests for the sad path where you'll be testing how it behaves when you feed it with invalid values such as invalid command-line flags.

Creating a test environment type

Let's create a new test environment type before testing the `run` function in Listing 5.21. The test environment type will run the tool in a controlled test environment. It will make it easier for you to write tests.

Listing 5.21: Creating a test environment type (hit_test.go)

```
package main

type testEnv struct {
   args           string              #A
   stdout, stderr bytes.Buffer        #B
}

func (e *testEnv) run() error {       #C
   s := flag.NewFlagSet("hit", flag.ContinueOnError)      #D
   s.SetOutput(&e.stderr)                                 #E
   return run(s, strings.Fields(e.args), &e.stdout)       #F
}
```

In Listing 5.21, the test environment type stores the necessary values, such as fake command-line arguments and standard output streams. The standard output streams become bytes buffers in the test environment type. Before running the run function, a test can change these values and provide a different testing environment.

Tip

`Buffer` implements the `Writer` interface. You can pass a buffer as a fake stream since the `run` function and flag set expect the same type.

There is also a `run` method in the test environment type that closely resonates with the `run` function. It creates a new unnamed flag set that continues on an error instead of terminating the program. It sets the output of the flag set to the fake standard error so that you can analyze it afterward and catch the errors yourself and investigate the standard output streams.

Note

The `NewFlagSet` function can create and return a new flag set. The flag package uses the default flag set called `CommandLine`. It uses the `NewFlagSet` function behind the scenes to create the default one.

Finally, Listing 5.21 runs the `run` function and passes the fake flag set, flags, and standard output streams. Then it returns an error from it. Since the `run` function expects a string slice, the `run` method splits the fake command-line arguments by space using the `Fields` function before running the `run` function. Passing the arguments as a single string value will make it easy to work within tests.

Tip

You could also store other fake values such as fake environment variables in the test environment and provide them to the `run` function. You could use the `os.Environ` function to do that.

Testing the happy path

You're now ready to test the happy path. You'll create two test cases:

- Passing a proper `url` flag.
- Passing proper `n` and `c` flags.

Each test case will describe what it expects. Then you'll test each test case to see there is no error and the fake standard output stream contains a valid output (the tool's plan message). Let's get to it in Listing 5.22.

Listing 5.22: Testing the happy path (hit_test.go)

```
func TestRun(t *testing.T) {
   t.Parallel()    #A

   happy := map[string]struct{ in, out string }{          #C
       "url": {
           "http://foo",
           "100 requests to http://foo with a concurrency level o
```

```
                strconv.Itoa(runtime.NumCPU()),
        },
        "n_c": {
            "-n=20 -c=5 http://foo",
            "20 requests to http://foo with a concurrency level of
        },
    }
    for name, tt := range happy {
        tt := tt
        t.Run(name, func(t *testing.T) {
            t.Parallel()      #B

            e := &testEnv{args: tt.in} #D
            if err := e.run(); err != nil {      #E
                t.Fatalf("got %q;\nwant nil err", err)
            }
            if out := e.stdout.String(); !strings.Contains(out, tt
                t.Errorf("got:\n%s\nwant %q", out, tt.out)
            }
        })
    }
}
```

Warning

The line `tt := tt` in Listing 5.22 ensures that each subtest gets a fresh copy of a test case. Otherwise, the subtest closure would *capture* the variable's memory address *once*, and the remaining subtests would see the same test case when they run.

The `t.Parallel` method marks a test as a parallel test, and the test package will run the test in parallel to other tests, if any. There are two calls to the `Parallel` method in Listing 5.22. The first one makes the `TestRun` run parallel to the other top-level tests. And the second one marks each subtest to run parallel to other tests. If you didn't call it from the subtests, they would not run in parallel.

Tip

Running tests in parallel make them run faster and finish in less time. However, you need to make sure that each test does not mess with globals, or you will face concurrency issues.

Listing 5.22 declares a new test function called `TestRun` to test the happy path. The same test will also include the sad path in a minute. The `happy` variable contains the happy path test cases in a `map`, and each test case is a `struct`. The `struct` has only two fields: `in` and `out`. The `in` field is fake command-line arguments that the test will pass to the `run` function. And the `out` field is what the test case expects when the `run` function finishes running.

Each subtest ensures that the test environment belongs only to a single test case so that there won't be any concurrency issues. If the test cases were sharing a single test environment, it would practically invalidate what you've been trying to do. Remember, your goal is to make the `main` function run in an isolated environment. That's why each subtest creates a new test environment.

The subtest finally calls the `run` method and runs the `run` function in an isolated test environment with fake command-line arguments and standard output streams. It ensures that there is no error and the standard output contains a valid message such as: `"20 requests to http://foo with a concurrency level of 5"`.

Note

You can write into a bytes buffer, and it will return what you wrote when you call its `String` method. The `run` function sees the fake bytes buffer as the standard output without knowing. So it writes to the bytes buffer then you get the data using the `String` method.

The test will pass when you run it:

```
$ go test.
ok ...
```

Congrats!

Testing the sad path

It's time to test the sad path in Listing 5.23. You'll create many test cases as testing for the edge cases is often exhaustive. You'll check how the `run` function behaves when you feed it with invalid flags. You'll be checking the standard error stream because the `run` function writes the error and usage messages if there is an error. You'll also ensure it returns with an error and that the standard error stream is not empty.

Listing 5.23: Testing the sad path (hit_test.go)

```
func TestRun(t *testing.T) {
    t.Parallel()
    ...
    happy := ...
    sad := map[string]string{
        "url/missing": "",
        "url/err":     "://foo",
        "url/host":    "http://",
        "url/scheme":  "ftp://",
        "c/err":       "-c=x http://foo",
        "n/err":       "-n=x http://foo",
        "c/neg":       "-c=-1 http://foo",
        "n/neg":       "-n=-1 http://foo",
        "c/zero":      "-c=0 http://foo",
        "n/zero":      "-n=0 http://foo",
        "c/greater":   "-n=1 -c=2 http://foo",
    }
    for name, in := range happy { ... }
    for name, in := range sad {
        in := in
        t.Run(name, func(t *testing.T) {
            t.Parallel()

            e := &testEnv{args: in}
            if e.run() == nil {       #A
                t.Fatal("got nil; want err")
            }
            if e.stderr.Len() == 0 {  #B
                t.Fatal("stderr = 0 bytes; want >0")
            }
        })
    }
}
```

Listing 5.23 adds the sad path test cases to the same `TestRun` function and runs them after the `TestRun` function runs the happy test cases.

Tip

Keeping both the happy and sad paths in the same test function can allow you to easily pinpoint when you make a mistake. You can change the test cases and code in the same function instead of wondering in other places.

Each subtest will run the `run` function and ensure that there is an error. It will also check the standard error is not empty, and the `run` function and friends write something to it. Remember, the `run` function (hence the flag package) prints the error and usage message to the standard error.

Warning

The bytes buffer's `Len` method returns the number of bytes written. Since the `Len` method returns the unread portion of the buffer, the method will return something else if you read the buffer, though.

5.7.4 Using build tags

Sometimes your tests can get longer to run, and you don't want to run them.

You have two options when that happens:

- Fixing the tests and making them performant (!).
- Using the `testing.Short` function and the `t.SkipNow` method.
- Using a build tag.

`Short` will return `true` if you run tests as follows:

```
$ go test -short
```

You can check for it in the test:

```
func TestRun(t *testing.T) {
   if testing.Short() {
       t.SkipNow()
   }
   ...
}
```

The `SkipNow` function will skip running the `TestRun` test, but it wouldn't have stopped the other tests if there were.

Using the `Short` and `SkipNow` functions can sometimes be a hassle when you have many tests, as you'll need to check the `Short` function for each to prevent running them. You can use a build tag if that's the case.

For example, say you don't want to run the `TestRun` test until you pass a tag called `cli`. You can add a build tag to the top of the `hit_test.go` file to do that:

```
//go:build cli
```

Beware

Do not add an extra space character after the double slashes (`//`) at the beginning.

Now, the `go test` command won't run any tests in the test file. However, it will run them if you provide the build tag from the command line as follows:

```
$ go test -tags=cli
```

You could do the opposite and exclude any tests that have the build tag as follows:

```
//go:build !cli
```

This last one will make you run the `cli` tests by default. But the test package won't run them if you provide the flag, and it will continue running the other tests.

People widely use build tags in the wild to separate their unit tests from integration tests.

Tip

You can learn more about build tags at the link https://pkg.go.dev/cmd/go#hdr-Build_constraints

5.7.5 Wrap up

- Testing the main function should not be different than testing any other function.
- The main function is challenging to test because it often uses globals.
- Extracting code from the main function to another replica function and injecting fake values from outside makes it testable.
- The test environment type makes testing the main function with fake values.
- Running tests in parallel makes them run faster and finish in less time. However, you need to make sure that each test does not mess with a global value, or you will face concurrency issues.
- The `t.Parallel` method marks a test as a parallel test, and the test package will run the test in parallel to other tests, if any.
- The `testing.Short` function and the `t.Skip` method allows you to run a group of tests selectively.
- The build tags allow you to customize the compilation of your files. You can use them to exclude a group of tests from running.

5.8 Exercises

Make sure to add relevant fields (to the `flags` struct) and tests for the following exercises.

1. Add a new timeout flag using the `DurationVar` method:

```
$ go run . -t=5s http://foo
Making ... (Timeout=5s).
```

2. Add a new flag with a new *dynamic value type* that accepts only the following values: `GET`, `POST`, and `PUT`:

```
$ go run . -m=GET http://foo
Making 100 GET requests to...
$ go run . -m=FETCH http://foo
invalid value "FETCH" for flag -m: incorrect method: "FETCH"
```

Add a new flag with a new dynamic value type that puts HTTP headers into a slice:

```
$ go run . -H='Accept: text/json' -H='User-Agent: hit' http://foo
```

```
Headers: "Accept: text/json", "User-Agent: hit"
Making ...
```

Tip: The user passes the same flag twice, and the flag package calls the `Set` method of the dynamic value twice! You can use this fact and append the values to a slice in the dynamic type.

5.9 Summary

You started the chapter from zero and created and tested a command-line tool in the end. Congrats! In the next chapter, you'll learn how to integrate the hit package into the tool and make concurrent HTTP requests.

Let's see what you have learned so far:

- You learned how to structure code for a command-line tool.
- You first created a parser using the os package. Then you upgraded the tool using the flag package and let it parse and validate command-line flags.
- You learned how to add your own validator because the flag package does not support validating required and dependent flags.
- You learned the inner mechanics of the flag package and extended it by adding your value type by implementing the flag package's `Value` interface.
- Finally, you learned how to make an untestable function testable by extracting the main function to another replica function. On top of that, you also learned how to create a test environment for the tool.

6 Concurrent API Design

This chapter covers

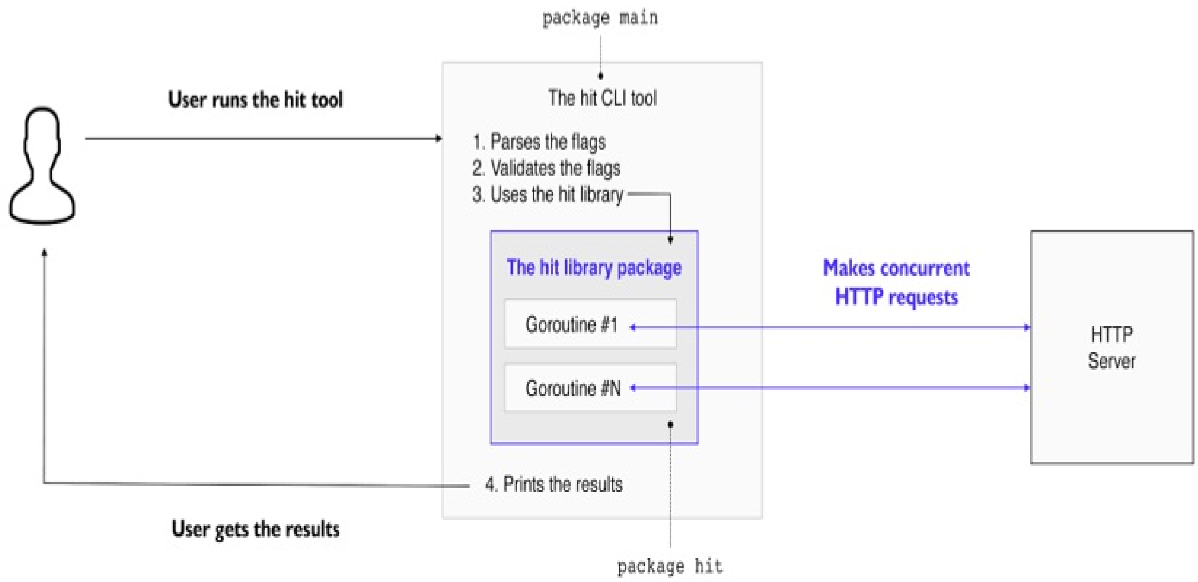
- Designing, implementing, testing, and optimizing a concurrent HTTP client.
- Learning about the http and httptest packages.
- Using the context and signal packages for cancellation.
- Implementing Rob Pike's Self-Referential Option Functions pattern.

Between 1989 and 1991, Tim Berners-Lee and his team at CERN were working on a protocol called HTTP that would be the enabler of the World Wide Web we have today. The protocol was based on exchanging *plain text messages* between a server and client, where the client sends a request with a simple text message, and the server returns a response body.

The last chapter taught you to create a command-line tool called *hit* that parses and validates command-line flags. This chapter will teach you to write *an idiomatic HTTP client package called the hit library* and integrate it into the *hit tool*. The package will send concurrent requests to a server to squeeze every bit of juice out of it to give general info to users about the server's performance.

Imagine you want to send thousands of requests to a server to analyze its performance using the *hit tool*. In Figure 6.1, the tool parses and validates the command-line flags and imports the *hit library*. The library sends concurrent requests, collects results, and returns an aggregated result to the tool. Finally, the tool shows the aggregated result to the user.

Figure 6.1 The hit project's overall architecture.



The first section will detail the library's architecture to help you design and implement the library. The next section will teach you the concurrent pipeline pattern to neatly structure concurrent code. Then, you will learn about *cancellation* to cancel ongoing requests and still get the results up to that moment. After that, you'll learn how to use and optimize the HTTP package. The chapter will finish after you successfully tested and refactored the library.

Let’s get started!

Warning

Please read Appendix A before starting the chapter if you are not adept at concurrency in Go.

6.1 Designing an idiomatic HTTP client

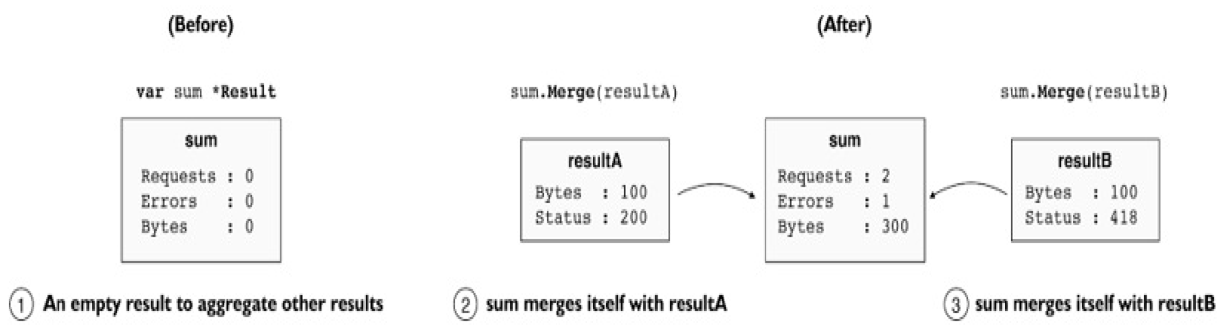
Achieving an effective architecture is mostly about reducing complexity by dividing a task into composable parts where each will be responsible for doing a smaller set of tasks. This section will teach you the hit library's architecture and implementation.

The library has two main types: `Client` and `Result`.

- `Client` orchestrates sending concurrent requests and collecting their results into a single aggregated `Result`.
- `Result` is a request's result, such as duration, status, etc. The type also lets you *merge* request results to see the server's overall performance as an aggregated result.

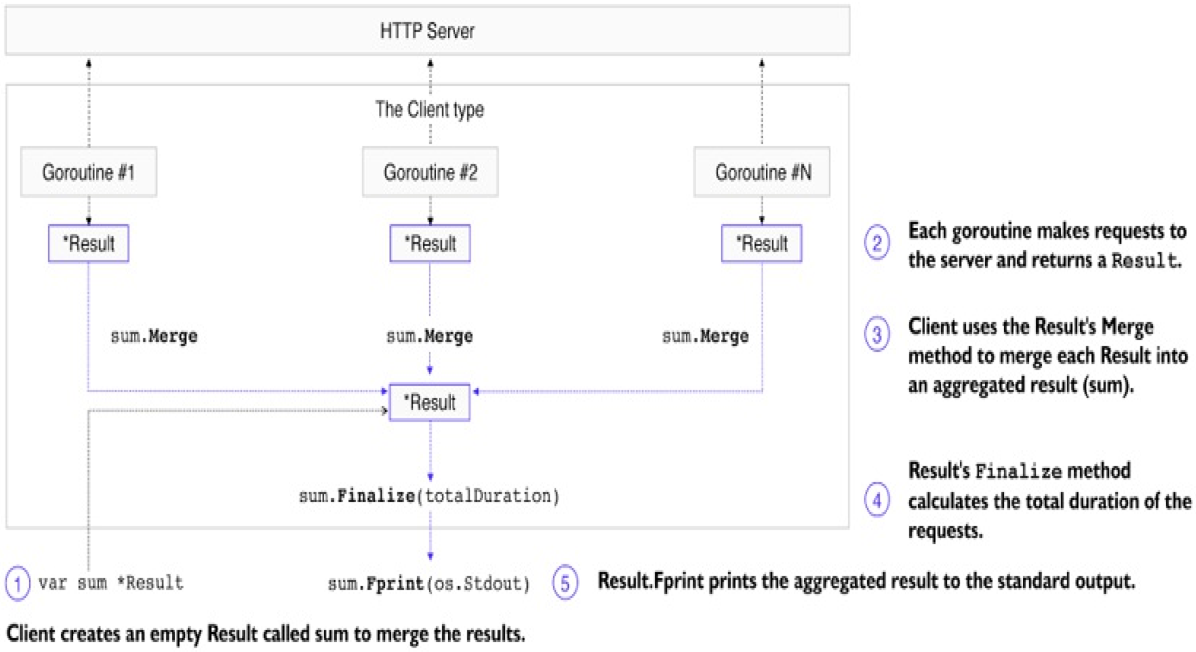
Imagine sending *two requests* to a server. You can `Merge` them into a single one to see the server's overall performance. Figure 6.2 creates an *empty result* to aggregate the results (`sum`). Then, it *merges* the results. Since the second result has an HTTP error code (`418=Teapot`), the `Errors` field becomes one.

Figure 6.2 The `Merge` method merges two results into a single aggregated result.



Now that you understand the hit library's core types, let's discuss their behavior. As shown in Figure 6.3, the `Client` type sends concurrent requests, uses the `Result` type to store request results, aggregate them, and show the aggregated result to users.

Figure 6.3 `Client` sends concurrent requests and returns an aggregated `Result` by merging request results.



1. The `Client` type creates an empty `Result` to aggregate other results.
2. It sends concurrent requests to a server via goroutines and collects *results* from the goroutines (as `Result` values).
3. Then it aggregates each result using the `Merge` method while the requests are in progress.
4. Then it calculates the total duration using the result's `Finalize` method.
5. Finally, it returns an *aggregated result* to the hit tool. The hit tool calls the result's `Fprint` method to show the overall performance result.

Let's diffuse a possible foggy area about the difference between the `Merge` and `Finalize` methods. The `Merge` method takes a finished request result from a goroutine and updates the aggregated result's number of requests field, the number of bytes downloaded field, etc.

So far, so good. The subtle thing is there is also a total duration field on the aggregated result. The total duration field contains information about how long it takes to finish all the requests. However, the `Merge` method should not update the total duration field of the aggregated result. For that, the `Client` type calls the `Finalize` method once all the requests are finished.

So, why should the `Client` use the `Finalize` method instead of the `Merge` method to calculate the total duration? Imagine one of the requests takes one and the other two seconds. Since the goroutines could run in parallel, the requests could finish in two instead of three seconds. If `Merge` were to add each duration to the aggregated result, the total duration would be three seconds instead of two.

Since you learned about the overall architecture, let's implement the hit library and integrate it into the hit tool. It makes sense to begin with the `Result` type as the `Client` type cannot work without the `Result` type.

HTTP Status Code 418: Teapot

HTTP status code 418 (the error you see in Figure 6.2) started as an April's fools joke, a reference to Hyper Text Coffee Pot Control Protocol (https://datatracker.ietf.org/doc/html/rfc2324). People wanted to remove status code 418, and others started the "Save 418 Movement" (https://save418.com).

6.1.1 Storing performance results

You can use the `Result` type to store a request result, merge results into an aggregated one, and print it. In this section, you will create the hit library's directory, implement the `Result` type, and integrate the library into the hit tool.

Let's recall the hit project's directories:

```
.                       -> Current directory
└── cmd                 -> A directory for executables
│   └── hit             -> The hit tool's directory
└── hit                 -> The hit library's directory
```

Let's create the `hit` directory and add the `Result` type in a new file, as shown in Listing 6.1.

The `Result` type has some fields for tracking a request's result. Some fields are only for an aggregated result (*e.g.*, `Fastest` *and* `Slowest`), while others are for a single one (*e.g.*, `Error` *and* `Status`). And the rest are for both.

The `Merge` and `Finalize` methods are self-explanatory. `Merge` merges a result with another to create an aggregated result. `Finalize` sets the total duration and RPS (requests per second) for an aggregated result. The `Fprint` method prints a result using an interface type called `Writer` (line 44).

Tip

In a switch statement, "`fallthrough`" runs the next case clause without checking the case's condition. Since in either situation (if there was an error or the status code was equal or above bad request), you would increment the number of errors.

The `round` and `success` are helpers for `Fprint`. Between lines 45 and 47, `p` closure keeps the `Fprint` method concise. Like the `Fprintf` function, `p` takes a *format string* and *variadic* input values. Each variadic input value's type is `any`, meaning it can be of any type of value. `args...` unpacks the variadic input values and passes each to `Fprintf`. In my article, you can learn more about variadic functions: https://blog.learngoprogramming.com/golang-variadic-funcs-how-to-patterns-369408f19085

Listing 6.1: Implementing the Result type (./hit/result.go)

```
package hit    #A

// Result is a request's result.
type Result struct {
    RPS      float64       // RPS is the requests per second
    Requests int           // Requests is the number of requests
    Errors   int           // Errors is the number of errors occu
    Bytes    int64         // Bytes is the number of bytes downlo
    Duration time.Duration // Duration is a single or all request
    Fastest  time.Duration // Fastest request result duration amo
    Slowest  time.Duration // Slowest request result duration amo
    Status   int           // Status is a request's HTTP status c
    Error    error         // Error is not nil if the request is
}

// Merge this Result with another.
func (r *Result) Merge(o *Result) {
    r.Requests++
    r.Bytes += o.Bytes

    if r.Fastest == 0 || o.Duration < r.Fastest {
```

```
        r.Fastest = o.Duration
    }
    if o.Duration > r.Slowest {
        r.Slowest = o.Duration
    }

    switch {
    case o.Error != nil:
        fallthrough    #B
    case o.Status >= http.StatusBadRequest:
        r.Errors++    #B
    }
}

// Finalize the total duration and calculate RPS.
func (r *Result) Finalize(total time.Duration) *Result {
    r.Duration = total
    r.RPS = float64(r.Requests) / total.Seconds()
    return r
}

// Fprint the result to an io.Writer.
func (r *Result) Fprint(out io.Writer) {
    p := func(format string, args ...any) {    #C
        fmt.Fprintf(out, format, args...)    #C
    }    #C
    p("\nSummary:\n")
    p("\tSuccess    : %.0f%%\n", r.success())
    p("\tRPS        : %.1f\n", r.RPS)
    p("\tRequests   : %d\n", r.Requests)
    p("\tErrors     : %d\n", r.Errors)
    p("\tBytes      : %d\n", r.Bytes)
    p("\tDuration   : %s\n", round(r.Duration))
    if r.Requests > 1 {
        p("\tFastest    : %s\n", round(r.Fastest))
        p("\tSlowest    : %s\n", round(r.Slowest))
    }
}

func (r *Result) success() float64 {    #D
    rr, e := float64(r.Requests), float64(r.Errors)    #D
    return (rr - e) / rr * 100    #D
}

func round(t time.Duration) time.Duration {    #E
    return t.Round(time.Microsecond)    #E
}    #E
```

The `Result` type exports all of its fields and does not use *getters and setters*. Most gophers (including me) think getters and setters come with problems, and we don't use them unless they are critical. For example, you could use getters and setters if changing an exported field can cause trouble in the inner workings of a type.

The `round` function is not a method, while the success is. It's because `round` doesn't use the `Result` type's fields while the latter does. It's better to declare a method when you need the fields (a.k.a. state); otherwise, declare a function.

The `Fprint` method takes an interface to make the method easy to test and reusable. For example, you could save the result into a file if you pass a `File` since `File` has a `Write` method (see https://pkg.go.dev/io#Writer and https://pkg.go.dev/os#File for more details).

Let's hit the road

Let's take the `Result` type for a ride. Since you have created the hit library, you can now import it from the hit tool. *You will continue from where you left off in the last chapter's code.* Let's go to the hit tool's directory (`./cmd/hit`) and update `hit.go` as in Listing 6.2.

Warning: Using your Go module

Listing 6.2 imports the hit library from the book's repository (see line 4). You should use your module path if you're not using the book's repository. For more information, you may look at the About Go modules notice in Section 5.2.1.

If your directory structure looks like the following:

```
project_name     -> Your project directory
    └── cmd
    │   └── hit      -> The hit tool's directory
    └── hit          -> The hit library's directory
```

Then you should import the hit library from the hit tool as follows:

```
import "github.com/username/project_name/hit"
```

Suppose there are three requests, and each of the first two requests takes one second, and the last one takes two seconds, and they both finish in two seconds since they might have run *in parallel*! Although the total duration of requests is four seconds (lines 12, 21, and 25 in Listing 6.2), `Finalize` pretends they were parallel and finished in two seconds (line 27).

Listing 6.2: Using Result (./cmd/hit/hit.go)

```
...
import (
    ...
    "github.com/inancgumus/effective-go/ch06/hit"    #A
)

func run(s *flag.FlagSet, args []string, out io.Writer) error {
    ...
    fmt.Fprintf(out, "Making %d requests to %s with a concurrency
        f.n, f.url, f.c)

    var sum hit.Result
    sum.Merge(&hit.Result{
        Bytes:    1000,
        Status:   http.StatusOK,     // Status Code=200 (not an e
        Duration: time.Second,
    })
    sum.Merge(&hit.Result{
        Bytes:    1000,
        Status:   http.StatusOK,
        Duration: time.Second,
    })
    sum.Merge(&hit.Result{
        Status:   http.StatusTeapot, // Status Code=418 (error)
        Duration: 2 * time.Second,
    })
    sum.Finalize(2 * time.Second)    // Assumes both requests too
    sum.Fprint(out)

    return nil
}
```

You pretend as if you send three requests to a server, then you receive and merge their results, calculate their total duration, and finally print an aggregated result to the console. Please run the following command while in the `./cmd/hit/` directory.

```
$ go run . http://localhost:9090
Summary:
        Success    : 67%
        RPS        : 1.5
        Requests   : 3
        Errors     : 1
        Bytes      : 2000
        Duration   : 2s
        Fastest    : 1s
        Slowest    : 2s
```

The requests seem to finish in two seconds because you set the total duration to two seconds in the `Finalize` method. RPS is one and a half because you made three requests in two seconds. You're one step closer to implementing the hit library. Exciting!

Bonus: Making the Result type a Stringer

Sometimes it can be more convenient to get a result as a string. Since most methods in the stdlib and other packages in the wild support the `String` method, you can provide a `String` method that uses the result's `Fprint` method.

The strings package's `Builder` type is a `Writer` (implements the `Write` method). Since the `Fprint` method can write to a writer, you can write to a `Builder`. The following `String` method writes the result to a buffer using the strings package's `Builder` type, and returns the buffer's content as a string:

```
func (r *Result) String() string {
    var s strings.Builder
    r.Fprint(&s)
    return s.String()
}
```

The fmt package's printing methods can recognize if a value you want to print is a `Stringer` (implements the `String` method) and seamlessly call the method. For example, you can now print the result as follows (you can try it in the hit tool by replacing the `Fprint` method with the following (see Listing

6.2):

```
fmt.Fprint(out, sum)
```

You could also print the result using other printing functions such as `Print` as follows:

```
fmt.Print(sum)
```

You won't use the result's `String` method in the rest of the chapter; it's good to learn what the `Fprint` method is capable of.

Wrap up

Let's summarize what you have learned in this section.

- The hit library has two main types: `Client` and `Result`.
- `Result` is a request's result, allowing you to merge other results and print. The `Merge` method aggregates results, `Finalize` sets the total duration, and `Fprintf` prints the result to the console (or to any `io.Writer`!).
- The `Client` type uses the `Result` type to store request results, aggregate them (`Merge` and `Finalize`), and print the aggregated result (`Fprintf`).
- Getters and setters come with problems, and we use them only when necessary.
- Declare a method when you need a state; otherwise, declare a function.

6.1.2 Designing and implementing the Client type

This section will teach you to design and implement an effective, easy-to-use, and maintainable HTTP client type. You'll implement the client type in the hit library and integrate it into the hit tool at the end of the section.

How to design a better API

The more critical thing to plan for is not only the package's internals but its externals first (its API). API is what you export from a package. Providing a straightforward API is critical as it's the only part the package's importers depend on and see.

Tip

A simple API is better than a complicated one. Hide complexity behind a simple API.

Here are the guidelines when creating an idiomatic API:

- Easy to use and not confusing for its users.
- Hides internal complexity by providing a simple API on the surface.
- Consists of composable parts and allows users to bring them together in ways its creator could not imagine.
- Synchronous by default.
- Do not create unnecessary abstractions.
- Trust programmers and don't babysit them. Have them control and fine-tune the API behavior however they need.

As you progress in the chapter, you'll understand these guidelines better.

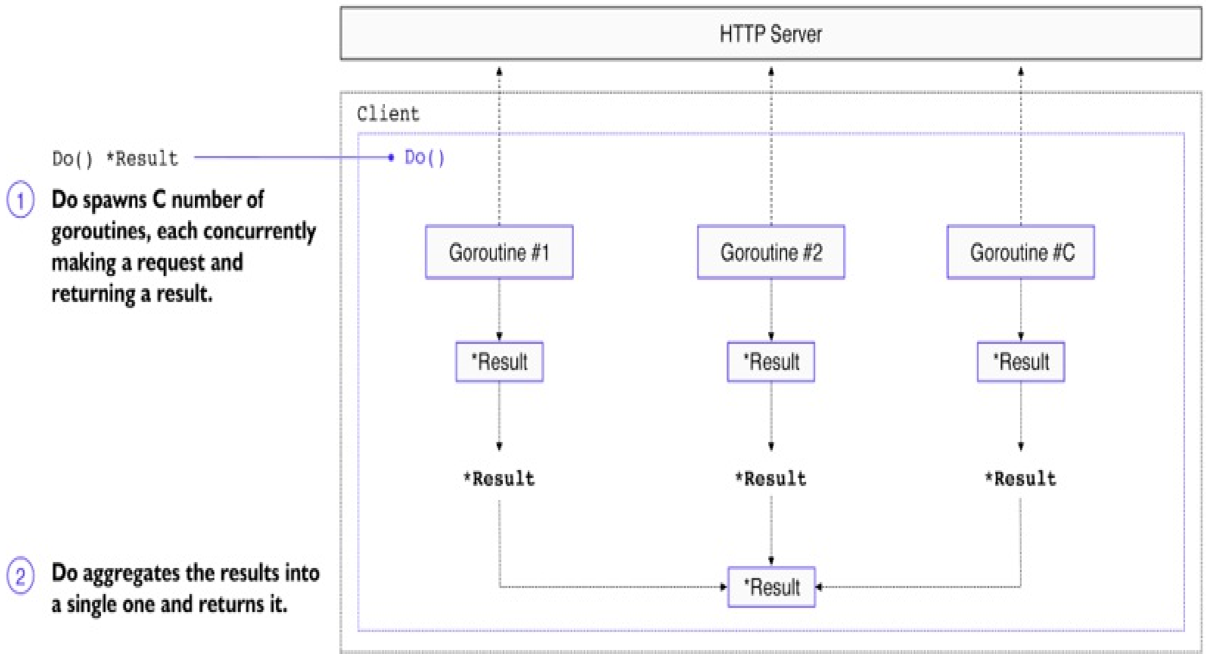
Learning about the Client type

Let's start discussing how the client type works. As shown in Figure 6.4, the hit library's `Client` type is an HTTP client that orchestrates sending concurrent requests to the same URL and returning an aggregated result.

Even though the `Client` has internal concurrent parts, its API is synchronous on the surface:

1. The `Do` method, when called, internally launches goroutines, each concurrently making a request and returning a result to the method.
2. Then, the `Do` method merges the results into an aggregated result and returns the result to the caller.

Figure 6.4 The `Do` method sends concurrent requests, merges the results into an aggregated result, and returns the aggregated result.



The `Do` method is synchronous and never returns until it makes all the requests:cl

- Does not accept a channel argument (or return one) to communicate with its caller.
- Leaves the decision of calling it asynchronously (in a goroutine) or not to its callers.

For example, imagine you want to send requests to multiple URLs concurrently. To do so, you can launch multiple goroutines, each calling the `Do` method.

Tip

Concurrency is an implementation detail. Design a synchronous API and let the users of your package decide when to use concurrency or not. For example, the Go standard library's API is mostly synchronous.

Implementing the Client type

Now that you understand the high-level design, let's implement the basic structure in Listing 6.3. You will also create a function called `Send` in Listing 6.4 to imitate sending an HTTP request and call the `Send` function from the `Client`.

Tip

It's a common practice to create twins. `Do` is like an entry point and handles higher-level stuff, while `do` handles the specifics.

The `Client` type has two methods called `Do` and `do`. The first one will call the `do` method and measure the total duration of the requests. While the latter will send HTTP requests, merge each request's performance result, and return an aggregated result. Both take the http package's `Request`.

Note

The `Request` type carries information such as which URL to send a request, HTTP headers, and cookies to use while sending the request. https://pkg.go.dev/net/http#Request.

Listing 6.3: Implementing the Client type (./hit/client.go) package hit

```
import (
    "net/http"
    "time"
)

// Client sends HTTP requests and returns an aggregated performan
// result. The fields should not be changed after initializing.
type Client struct {
    // To be added later.
}

// Do sends n HTTP requests and returns an aggregated result.
func (c *Client) Do(r *http.Request, n int) *Result {
    t := time.Now()    #A
    sum := c.do(r, n)
    return sum.Finalize(time.Since(t))    #A
}

func (c *Client) do(r *http.Request, n int) *Result {
```

```
    var sum Result
    for ; n > 0; n-- {
        sum.Merge(Send(r))    #B
    }
    return &sum
}
```

Note

Since the `Client` will work concurrently, it is critical to document that its fields should not be changed after setting them (line 9 in Listing 6.3). Otherwise, there can be concurrency bugs.

The `Client` type provides a simple API to send requests to a server and return an aggregated performance result. For now, you implemented the `Do` method without concurrency and will make it concurrent in the later sections.

Tip

`Since` returns the time duration between time values. It's a shortcut for `time.Now.Sub(t)`.

The `Send` function in Listing 6.4 takes a request value and uses it for printing the URL, but it has not yet made any real request to an HTTP server. The returned result imitates a successful request using the http package's `StatusOK` constant, which equals `200`.

Listing 6.4: The Send function to imitate requests (./hit/request.go) package hit

```
import (
    "fmt"
    "net/http"
    "time"
)

// SendFunc is the type of the function called by Client.Do
// to send an HTTP request and return a performance result.
type SendFunc func(*http.Request) *Result

// Send an HTTP request and return a performance result.
func Send(r *http.Request) *Result {
    t := time.Now()    #A
```

```
    fmt.Printf("request: %s\n", r.URL)
    time.Sleep(100 * time.Millisecond)    #B

    return &Result{
        Duration: time.Since(t),    #A
        Bytes:    10,
        Status:   http.StatusOK,    #C
    }
}
```

The `Send` function doesn't do anything useful yet and sends a fake request and returns a fake result. Then again, that will allow you to see the `Client` type in action. The `SendFunc` on line 11 is the type (signature) of the `Send` function, which you'll use to simplify passing the `Send` function as a *function value* later in the chapter (in Section 6.2.4).

Tip

Why export the `Send` function when only the `do` is using it? You do it to let the library's importers build their custom clients (as you did by creating the `Client` type).

Integrating the Client type into the hit tool

It's time to integrate the client into the hit tool and see it in action! Doing so will help you keep improving the client throughout the chapter. Listing 6.5 replaces the previous code you added to the hit tool's `run` function. The `run` function passes a new request to the client to send multiple HTTP requests to a server and prints the aggregated performance result.

Listing 6.5: Using Result (./cmd/hit/hit.go)

```
...
import (
    ...
    "net/http"
    ...
)
...
func main() {
```

```
    if err := run(flag.CommandLine, os.Args[1:], os.Stdout); err
        fmt.Fprintln(os.Stderr, "error occurred:", err)    #A
        os.Exit(1)
    }
}

func run(s *flag.FlagSet, args []string, out io.Writer) error {
    ...
    fmt.Fprintf(out, "Making %d requests to %s with a concurrency
        f.n, f.url, f.c)

    request, err := http.NewRequest(http.MethodGet, f.url, http.N
    if err != nil {
        return err    #A
    }
    var c hit.Client
    sum := c.Do(request, f.n)    #C
    sum.Fprint(out)    #C

    return nil
}
```

In Listing 6.5, you first create a new `GET` request that doesn't have a body (`NoBody`) with the URL you get from the command line. The `NewRequest` function returns an error if the provided values are faulty (line 22). The `main` function will get the error and print it to the standard error stream (line 10). If it's okay, you pass it to the `Do` method to send an HTTP request to a server.

Let's take it for a ride, shall we?

```
$ go run . -n 10 -c 1 http://localhost:9090
...
Summary:
        Success    : 100%
        RPS        : 9.9
        Requests   : 10
        Errors     : 0
        Bytes      : 100
        Duration   : 1.012753s
        Fastest    : 100.18ms
        Slowest    : 103.436ms
```

Above is what I see on my computer, but your mileage may vary. The tool made ten requests in a second, and RPS (requests per second) was about ten requests per second—*which is sluggish*. Next, let's see how you can make it faster by improving the client with concurrency.

6.1.3 Wrap up

You learned many tips and tricks, implemented the hit library package, and integrated it into the hit tool you developed in the last chapter. The current client doesn't make concurrent requests nor sends real HTTP requests. Then again, this section was a great first step towards these goals.

Let's quickly wrap up what you have learned in this section.

- `Client` sends requests and returns an aggregated result (as a `Result`).
- `Result` is a request's result. The `Merge` method merges a result with another, `Finalize` sets a result's total duration, and `Fprint` prints the result to the console.
- Hide complexity behind a simple and synchronous API and let other people decide when to use your API concurrently. Concurrency is an implementation detail.
- The http package's `Request` type determines which URL to send a request, HTTP headers, and cookies to use while sending the request. You can make a new request value using the http package's `NewRequest` function.

6.2 Designing a concurrent pipeline

Sending many sequential requests to a server wouldn't reflect its real performance. One request could take seconds, while the server could simultaneously serve other requests. Previously, you ran the hit tool and made ten requests in a second (RPS=10), but each was made sequentially. If you were to use ten goroutines, the RPS would be one hundred instead!

You can use concurrency to send requests faster since each would take more time than the communication time between goroutines (*about a microsecond or less*).

Let's design and implement a concurrent pipeline and integrate it with the client. Are you ready to squeeze the servers' last bit of performance?

Note

Go's definition of concurrency is, structuring a program as independently executing components. And the concurrent pipeline perfectly fits the bill. Think of a pipeline as a Unix pipeline or an assembly line in a car factory. https://go.dev/blog/pipelines.

6.2.1 What is a concurrent pipeline?

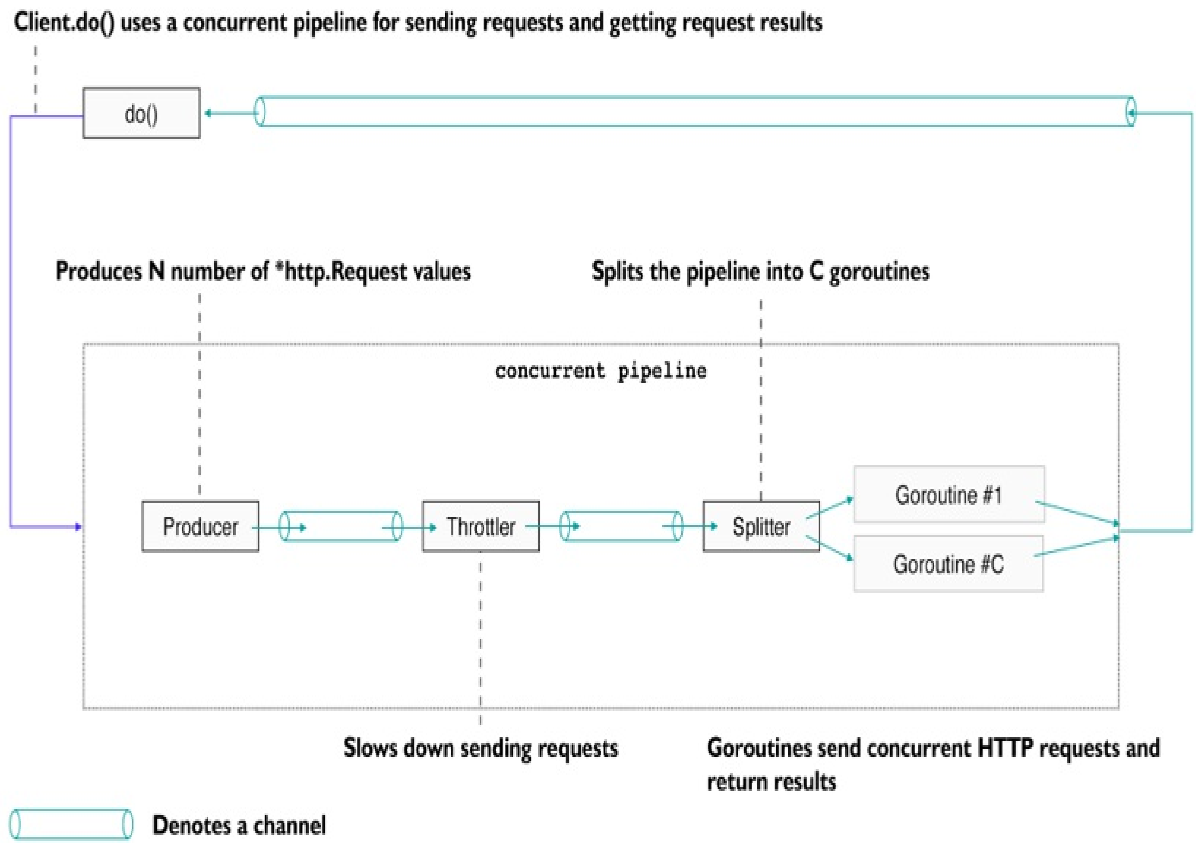
A pipeline is an extensible and efficient design pattern consisting of concurrent stages. Each stage does something and sends a message to the next via channels. The pipeline stages consist of *plain-old functions*, and the client's `do` method will connect them.

Note

The producer generates requests, but only the splitter sends them to a server.

- The **producer** generates requests, and other stages consume them.
- The **throttler** receives the requests and slows down the pipeline if desired—*hence slows down sending requests*. Then it sends the request to the next stage.
- The **splitter** runs goroutines to listen for the incoming requests.
- The goroutines send HTTP requests to a server and return results.

Figure 6.5 The concurrent pipeline's overall design. The `Client` type's `do` method makes a concurrent pipeline for sending requests and getting request results.

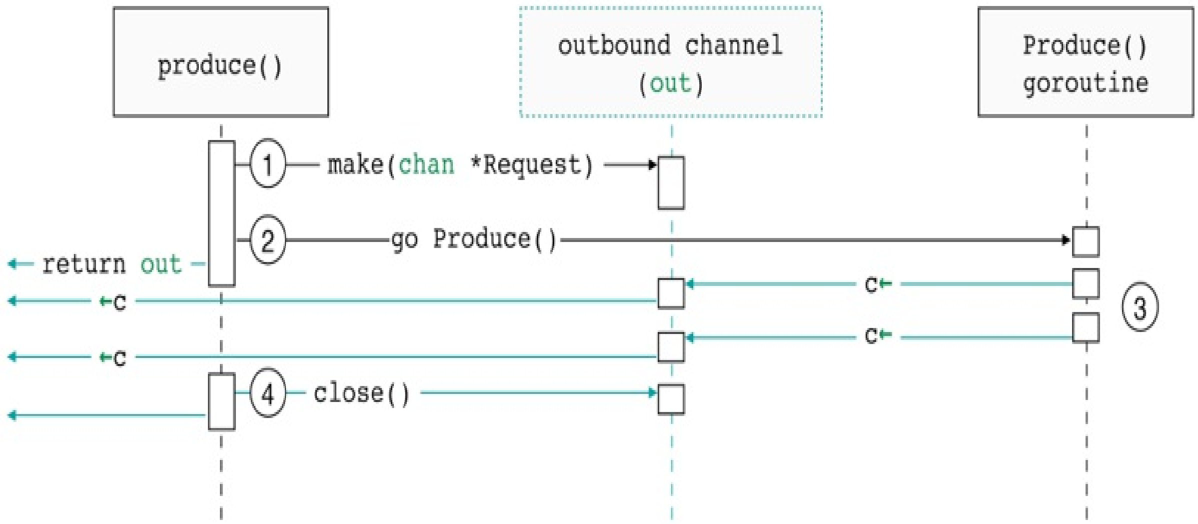


Since each stage communicates through channels, a different goroutine should run each stage. Otherwise, one stage would send a message to a channel where the next one listens, resulting in a deadlock unless you use buffered channels. If you were to use buffered channels, the stages wouldn't work in a lockstep manner and would lead to a hard-to-debug program. Let's keep it simple and effective.

6.2.2 Producer stage: Generating requests

The producer stage generates requests, and other stages consume them. Figure 6.6 details how the producer works. There will be two functions: The `produce` function (*the orchestrator*) runs the `Produce` (*the logic*) in a new goroutine.

Figure 6.6 The produce function creates a request channel, spawns a new Produce goroutine, and returns the channel to the next stage. The next stage listens to the channel for new request values until the produce function closes the channel.



1. The `produce` function creates and returns a channel to send the `Request` values.
2. The `Produce` function *runs in a new goroutine*.
3. The goroutine generates requests and sends each to the next stage via the channel.
4. The `produce` function *closes* the channel after the goroutine sends all the values. The rest of the stages will stop running when the producer closes the channel.

Since you understand that the producer generates requests and sends each to a channel, let's implement the producer. Let's add a new file (`pipe.go`) in the hit library's directory and then declare a new function called `Produce` in Listing 6.6.

Tip

Using directional channels (receive-only and send-only) prevents you from introducing bugs in the future and shows the intentions of what to do (and what you cannot and should not do) with a channel.

The `Produce` function generates requests and sends them to the output channel. It takes a *send-only* channel to send the generated requests. The second input determines how many request values the producer should generate. The last input is a function to leave the decision of how to produce requests to the caller. For example, you could add a unique identifier to each request for later analysis.

The unexported `produce` function runs the `Produce` function in a goroutine and returns a receive-only channel. The `produce` function first makes a channel (line 14) and then returns the channel (line 19) after starting the goroutine (line 15). It returns the channel while the goroutine keeps doing its job so the listener can receive values.

The goroutine will `close` the channel after delivering all the messages to tell the listener to stop listening. Otherwise, the caller would be blocked forever, waiting for more messages.

Listing 6.6: Implementing the producer (./hit/pipe.go)

```
package hit

import "net/http"

// Produce calls fn n times and sends results to out.
func Produce(out chan<- *http.Request, n int, fn func() *http.Req
    for ; n > 0; n-- {
        out <- fn()
    }
}

// produce runs Produce in a goroutine.
func produce(n int, fn func() *http.Request) <-chan *http.Request
    out := make(chan *http.Request)
    go func() {
        defer close(out)    #A
        Produce(out, n, fn)
    }()
    return out    #B
}
```

Tip

Only the channel's owner should close a channel to ensure all values are sent.

The producer is a simple function (`Produce`), but its effects are undeniable as it is the pipeline's entry point and enabler. The rest of the stages will listen for request values from the producer's output channel. They will stop working when the producer stops sending values to the channel (*by closing it*).

Previously, Section 6.1 suggested keeping concurrency as an implementation detail and not using channels in your public (exported) API. However, in this case (`Produce` in Listing 6.6), getting a channel from outside and having a concurrent API is fair. It's because the `Produce` function lets others build their pipeline (think of it like LEGO bricks). It is okay as long as it stays as a helper function and doesn't let others mess up your library's main logic.

The orchestrator function pattern

Note: I made up the name of this pattern. Then again, this pattern is widely used in the wild.

The `produce` function in Listing 6.6 is the orchestrator function that runs the producer's main logic (the `Produce` function) in a separate goroutine. Separating a function that contains the main logic (`Produce`) from the goroutine that would run it (`produce`) is a good pattern to follow. The `Produce` function in Listing 6.6 contains the core logic, but it doesn't matter whether you will run the producer in a new goroutine.

There is another benefit too. Instead of making a channel, the producer leaves the decision to the caller. So the caller can decide to create a buffered or an unbuffered channel to manage the stage more effectively if needed.

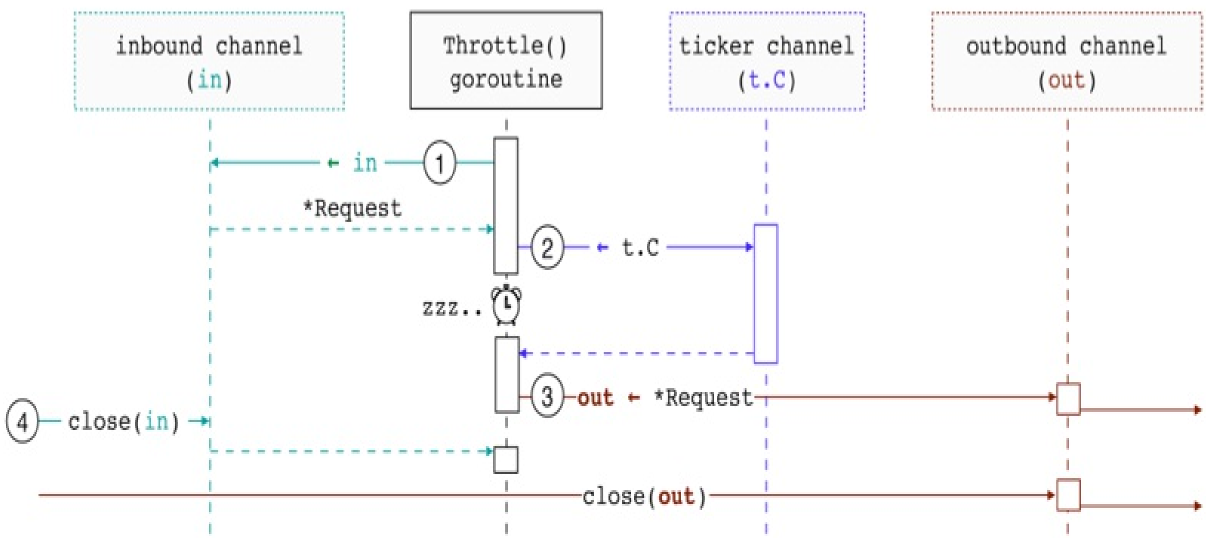
For example, the caller can use a buffered channel to buffer all the request values into the channel (hence into memory) upfront and immediately close the channel and free the producer. The producer's output channel would be filled with request values even if the producer is gone. And the rest of the stages would keep working until they are out of request values.

One last benefit is that the hit library's importers can create a custom client type. That's why you exported the `Produce` function so they can use it to create a pipeline if they want. But, the `produce` function is unexported because only the `Client`'s `do` method will use it.

6.2.3 Throttler stage: Slowing down the pipeline

When a server is overloaded with requests, you can't reliably see how it performs. This section will show how to slow down making requests (*throttling*) by adding a new stage called *throttler* into the concurrent pipeline. In Figure 6.7, the throttler receives reminders at periodic intervals from the stdlib's `Ticker` type to slow down delivering the request values.

Figure 6.7 The throttler delays requests values using a Ticker.



1. Throttler *receives* request values from the inbound channel `in` (producer).
2. Throttler *waits* for the *tick* (tick=ticker sends a message to the ticker channel `t.C`).
3. The throttler *wakes up* and *sends* the request to the outbound channel (`out`).
4. The throttler keeps looping (*steps 1-3*) until the inbound channel is closed, causing the throttler's orchestrator (*recall from the previous section about the orchestrator pattern*) to close the outbound channel so that the next stage will stop listening.

The throttler will receive request values from the producer when you connect the throttler stage to the pipeline. The throttler's inbound channel will be the producer's outbound channel. And the throttler's outbound channel will be the splitter’s inbound channel.

In Listing 6.7, the throttler (`Throttle`) takes a receive-only channel (*inbound*), a send-only channel (*outbound*), and a delay that specifies how long to wait before sending the request value to the outbound channel for each request value received from the inbound channel (line 2).

Note

The ticker's `C` field is a channel to block the current goroutine for a specified time (*delay*).

The throttler makes a ticker (line 4) to receive periodic ticks. The ticker periodically sends a message to a channel (`t.C`) that the throttler is waiting for (line 8). The throttler will stop the ticker as soon as the throttler gets all the values from the inbound channel (line 5).

Tip

`Ticker` will leak (consume system resources) a goroutine if you do not close it.

Listing 6.7: Implementing the throttler (./hit/pipe.go)

```
// Throttle slows down receiving from in by delay and
// sends what it receives from in to out.
func Throttle(in <-chan *http.Request, out chan<- *http.Request,
    t := time.NewTicker(delay)
    defer t.Stop()

    for r := range in {
        <-t.C
        out <- r
    }
}

// throttle runs Throttle in a goroutine.
func throttle(in <-chan *http.Request, delay time.Duration) <-cha
    out := make(chan *http.Request)
    go func() {
```

```
        defer close(out)
        Throttle(in, out, delay)
    }()
    return out
}
```

You used the pattern that you used with the producer stage. The `Throttle` contains the throttler stage's main logic, and the `throttle` runs the `Throttle` in a goroutine. Please review the producer stage if you don't remember about this pattern and its benefits.

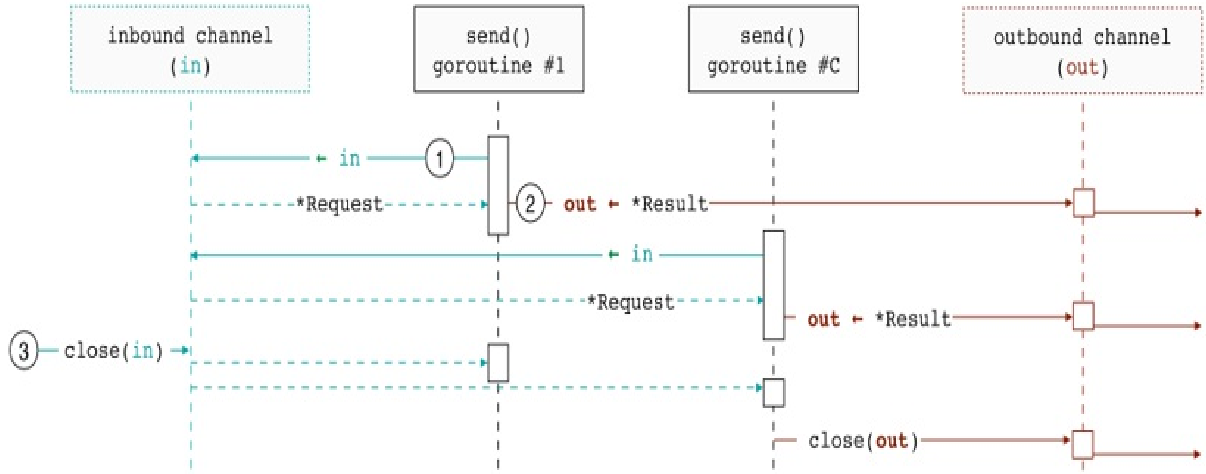
Note

Imagine you set the throttler to make one request every second, and the producer gets two seconds late sending messages to the throttler. If you were to use `Sleep`, the operation would take three seconds. However, `Ticker` would immediately tick since the producer was already late sending messages for two seconds.

6.2.4 Splitter stage: Sending parallel requests

As you saw in Section 6.1.2, sending requests to a server at a time is sluggish, and you can make it faster if you can send requests in parallel. The splitter stage splits the incoming request values among goroutines to send parallel requests (Figure 6.8).

Figure 6.8 Splitter spawns goroutines (two of them shown in the figure) which each receives an incoming request value, sends the request to a server, and sends the request result to the outbound channel.



1. Each goroutine receives a request from the inbound channel.
2. Each goroutine sends the request to a server, gets the result, and sends the result to the outbound channel.
3. The goroutines keep doing so (*steps 1-2*) until the inbound channel is closed, causing the goroutines' orchestrator to close the outbound channel.

Goroutines receive incoming request values from the inbound channel, send HTTP requests and get results, and then send the results to the outbound channel. Each goroutine will send `N/C` (the number of requests/concurrency level) requests to share the total amount of work.

In Listing 6.8, the splitter (`Split`) takes a receive-only channel to receive request values (`in`), a send-only channel to send request results (`out`), concurrency level (`c`—*which specifies the number of goroutines to spawn to process request values*), and a function that sends a request to an HTTP server and returns a result (`fn`).

Recall

In Section 6.1.2, the `SendFunc` type is declared as: `func(*http.Request) *Result`.

Splitter has a closure called `send` (lines 9-13) that receives request values from the inbound channel and sends the request results to the outbound channel using the given function (`fn`). Each goroutine will run the `send` function to send an HTTP request in parallel (lines 15-23).

Listing 6.8: Implementing the splitter (./hit/pipe.go)

```
// Split splits the pipeline into c goroutines, each running fn w
// what split receives from in, and sends results to out.
func Split(in <-chan *http.Request, out chan<- *Result, c int, fn
    send := func() {
        for r := range in { #A
            out <- fn(r)    #A
        }
    }

    var wg sync.WaitGroup
    wg.Add(c)
    for; c > 0; c-- {    #B
        go func() {    #B
            defer wg.Done()    #C
            send()
        }()    #B
    }    #B
    wg.Wait()    #D
}

// split runs Split in a goroutine.
func split(
    in <-chan *http.Request, c int, fn SendFunc,
) <-chan *Result {
    out := make(chan *Result)
    go func() {
        defer close(out)
        Split(in, out, c, fn)
    }()
    return out
}
```

Note

Please read Appendix A.7 for more information about WaitGroup.

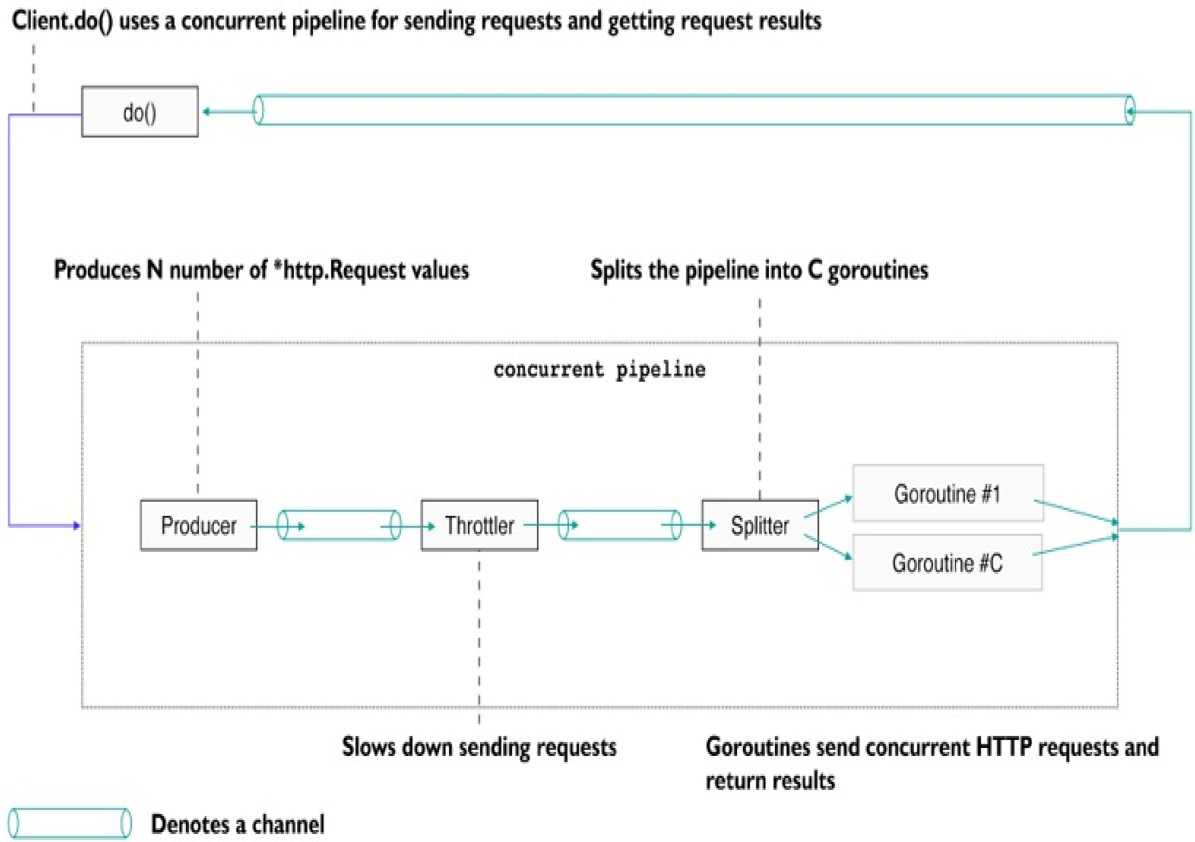
The splitter runs the given function in multiple goroutines to send parallel HTTP requests. The fourth input value, `fn`, has the same signature as the `Send` method you previously implemented in Listing 6.4. That will allow you to directly pass the `Send` function to the splitter when you're integrating the splitter into the pipeline.

6.2.5 Connecting the stages

The current client cannot send concurrent requests and is slow. Your goal has been to send requests faster by making them in parallel using a concurrent pipeline. Since you've finished implementing all the concurrent pipeline stages, let's discuss how you can connect them to the client and start sending parallel requests.

The client's `do` method connects the stages (Figure 6.9). *The producer* is the pipeline's starting point and generates new HTTP request values. *Throttler* slows down sending the request values to the next stage. You will activate the throttler stage *only when it's needed*.

Figure 6.9 The concurrent pipeline's overall design. The `Client` type's `do` method makes a concurrent pipeline for sending requests and getting request results.



This property is one of the beauties of using a pipeline that you can enable and disable individual stages depending on what you want to do with them. Each one is a concurrently executing component. Lastly, *Splitter* takes the request values from the *producer* **or** the *throttler* and spawns multiple goroutines to send HTTP requests in parallel.

Bringing the concurrent pipeline to life

Now that you understand how to connect the stages, let's implement the concurrent pipeline in Listing 6.9. How many request values the producer will produce depends on the `n` input value. And how many requests the splitter goroutines will send in parallel depends on the `C` field (line 3). Lastly, the `RPS` field throttles requests per second.

Tip

It's convenient to see what's happening in a method in one go: What the `do` method does is clear. Rather than burying the cloning of request values logic inside the `produce` function, the `produce` function takes a function and leaves the decision to the caller (in this case, the `do` method). You can also say the same for the `split` function. Plus, doing so allows you to change the `do` method's logic and the functions separately in the future. Lastly, the functions become more reusable (`produce` and `split`).

The action starts at the `Do` method. It takes a request value and passes it to the unexported `do` method. The callback you pass to the producer uses the request value *as a template* to produce new request values using the request value's `Clone` method (line 16).

Note

The callback clones the request to let the producer produce identical request values to send HTTP requests to the same URL. If request values were not cloned, there would be clashes with other ongoing request values since each request value is stateful.

In line 16, you get a `Context` value using the `TODO` function and pass it to the `Clone` method to clone the current request. The `TODO` function returns an *empty cancellation policy* –meaning it won't cancel anything. You use it here because you haven't updated the `Do` and `do` methods to get an upstream context. You will do so later in the chapter.

You want to enable throttling only when the user wants to throttle the requests; hence you'll do so if the `RPS` field is specified (lines 18-20). If throttling is enabled, you pass the producer's outbound channel as the inbound channel to the throttler and get the throttler's outbound channel (line 19). By doing so, the channel will become the throttler's inbound channel and the throttler will receive request values from the producer. Then, the loop will listen to the throttled channel instead (lines 22-24).

Tip

One of the strong aspects of using a concurrent pipeline is that you can easily add and remove stages and compose different pipelines without changing stage code.

The `do` function divides one second by `RPS` to determine how many requests to send per second (line 19). For example, the throttler will slow down each request value by half a second if `RPS` is two (`second/2=half a second`) so that the pipeline will send two requests per second. Then, you multiply the `RPS` field with the `C` field to adjust for concurrency.

Otherwise, the throttler would merely slow down all parallel requests to a level specified by the `RPS` field, and sending parallel requests wouldn't make sense. Lastly, you connect the splitter to the pipeline using the producer *or* the throttler's outbound channel (line 22).

The splitter will distribute the incoming request values to multiple goroutines depending on the `C` field. Each goroutine will run the `Send` function you developed earlier (in Listing 6.4 of Section 6.1.2) to send HTTP requests and get back performance results.

Each goroutine will send the performance results to the same channel that the splitter returns. Then you can listen for the new results until the splitter closes its outbound channel (line 22), merge each result into an aggregated result (line 23), and return the aggregated result (line 25).

Listing 6.9: Connecting the stages (./hit/client.go)

```
...
type Client struct {
    C   int // C is the concurrency level
    RPS int // RPS throttles the requests per second
}

// Do sends HTTP requests and returns an aggregated result.
func (c *Client) Do(r *http.Request, n int) *Result {
    t := time.Now()
    sum := c.do(r, n)
    return sum.Finalize(time.Since(t))
}

func (c *Client) do(r *http.Request, n int) *Result {
    p := produce(n, func() *http.Request {
        return r.Clone(context.TODO())
    })
    if c.RPS > 0 {
        p = throttle(p, time.Second/time.Duration(c.RPS*c.C))
```

```
    }
    var sum Result
    for result := range split(p, c.C, Send) {
        sum.Merge(result)
    }
    return &sum
}
```

The rest of the code is similar to the previous one you implemented in Listing 6.3 of Section 6.1.2. The difference is that you now use a concurrent pipeline and listen to the results from a channel to merge them to return an aggregated result. You connected all the stages and created a concurrent pipeline to process requests. Next, let's discuss the throttling flag.

Adding the throttling flag

It would help if you could get a throttling flag from the command line and set the `RPS` field. Otherwise, the throttler stage won't ever be active. Let the users decide how long to wait between requests by providing *a throttling flag* from the command line.

Listing 6.10 adds a new field called `rps` (line 3) (*requests per second to be used for throttling*). And it defines a new flag called `t` that only accepts positive numbers (line 9).

Remember

`toNumber` helps make a flag that accepts only natural numbers (`n>0`). Please review Section 5.5 in the previous chapter if you don't remember how the function works.

Listing 6.10: Adding the throttle field (./cmd/hit/flags.go)

```
type flags struct {
    url       string
    n, c, rps int
}

func (f *flags) parse(s *flag.FlagSet, args []string) error {
    ...
```

```
    s.Var(toNumber(&f.c), "c", "Concurrency level")
    s.Var(toNumber(&f.rps), "t", "Throttle requests per second")
    ...
}
```

Since you added a new flag called `t` that only accepts natural numbers, the users can provide requests per second (RPS) value from the command line. One thing left to do: Set the client's RPS field (line 11 in Listing 6.11) so the client can enable throttling in the concurrent pipeline (line 19 in Listing 6.9).

It would be better to tell the users that the hit tool is running in the throttled mode to see why the requests are throttled (lines 5-7 in Listing 6.11).

Listing 6.11: Integrating the throttling (./cmd/hit/hit.go)

```
func run(s *flag.FlagSet, args []string, out io.Writer) error {
    ...
    fmt.Fprintf(out, "Making %d requests to %s with a concurrency
        f.n, f.url, f.c)
    if f.rps > 0 {
        fmt.Fprintf(out, "(RPS: %d)\n", f.rps)
    }
    ...
    c := &hit.Client{
        ...
        RPS: f.rps,
    }
    sum, err := c.Do(request, f.n)
    ...
}
```

That's all! The users can now throttle the requests and see whether the hit tool runs in the throttled mode. Finally, let's take the concurrent pipeline for a ride and see how it behaves.

Taking the concurrent pipeline for a ride

You connected the stages to make a concurrent pipeline and added the throttling flag. Everything is ready for sending HTTP requests in parallel. It's time to try the hit tool and send requests in parallel.

Recall

It might be a good time to remember that the `Send` function imitates as if a request takes one hundred milliseconds (Listing 6.4 of Section 6.1.2).

The following command will send one thousand concurrent requests by distributing them among ten goroutines:

```
$ go run . -n 1000 -c 10 http://localhost:9090
...
Summary:
        Success    : 100%
        RPS        : 98.9
        Requests   : 1000
        Errors     : 0
        Bytes      : 10000
        Duration   : 10.110839s
        Fastest    : 100.008ms
        Slowest    : 104.565ms
```

It takes about ten seconds to send one thousand concurrent requests using ten goroutines on my machine. Let's try the same command and send sequential requests (that is, only with one goroutine):

```
$ go run . -n 1000 -c 1 http://localhost:9090
        RPS        : 9.9
        Duration   : 1m41.151939s
```

It takes about two minutes to send one thousand requests using a single goroutine on my machine. The performance difference is undeniable. The concurrent version is 10X faster! The throttling wasn't active in the previous runs.

Let's use the throttler and send concurrent requests with following command:

```
$ go run . -n 1000 -c 10 -t 1 http://localhost:9090
(RPS: 10)
        RPS        : 10
        Duration   : 1m40s
```

You told the hit tool to limit the requests per second to one. Since there were ten goroutines, the client adjusted the throttling to ten. As I explained in *Section 6.2.5's connecting the stages* subsection, otherwise, the throttler would merely slow down the requests to a level specified by the `RPS` field (*the* *hit tool would make one request per second*), and sending parallel requests wouldn't make sense. If you were not using the concurrency flag and set it to one RPS would be one too, and the total duration would be almost like forever.

6.2.6 Wrap up

You made it this far. Congrats! You started the section with a client that could only send sequential requests, and made it concurrent. You now have a tool (and a library) to send and process concurrent requests. You've added the concurrency to the hit library without changing its *observed behavior* (API).

Since *concurrency is an implementation detail*, the hit library users wouldn't even be aware that you made the library better.

Let's summarize:

- Go's definition of concurrency is, structuring a program as independently executing components. And the concurrent pipeline perfectly fits the bill.
- A pipeline is an extensible and efficient design pattern consisting of concurrent stages. You can easily add and remove stages and compose different pipelines without changing stage code.
- The producer stage generates requests, and other stages consume them. The throttler stage receives the requests and slows down the pipeline if desired—*hence slows down sending requests*. The splitter stage sends HTTP requests in parallel using goroutines.
- The `Ticker` type sends periodic intervals to a channel.
- The `Context` type's `TODO` function returns an uncancellable context you can use when you don't know what context value to pass to another function that accepts a context.

6.3 Graceful cancellation

Warning

Please read Appendix A if you don't know how the context package works.

Imagine you want to send millions of requests and, for some reason, want to cancel the ongoing work. Or, you might want to stop sending requests after a specific time (timeout).

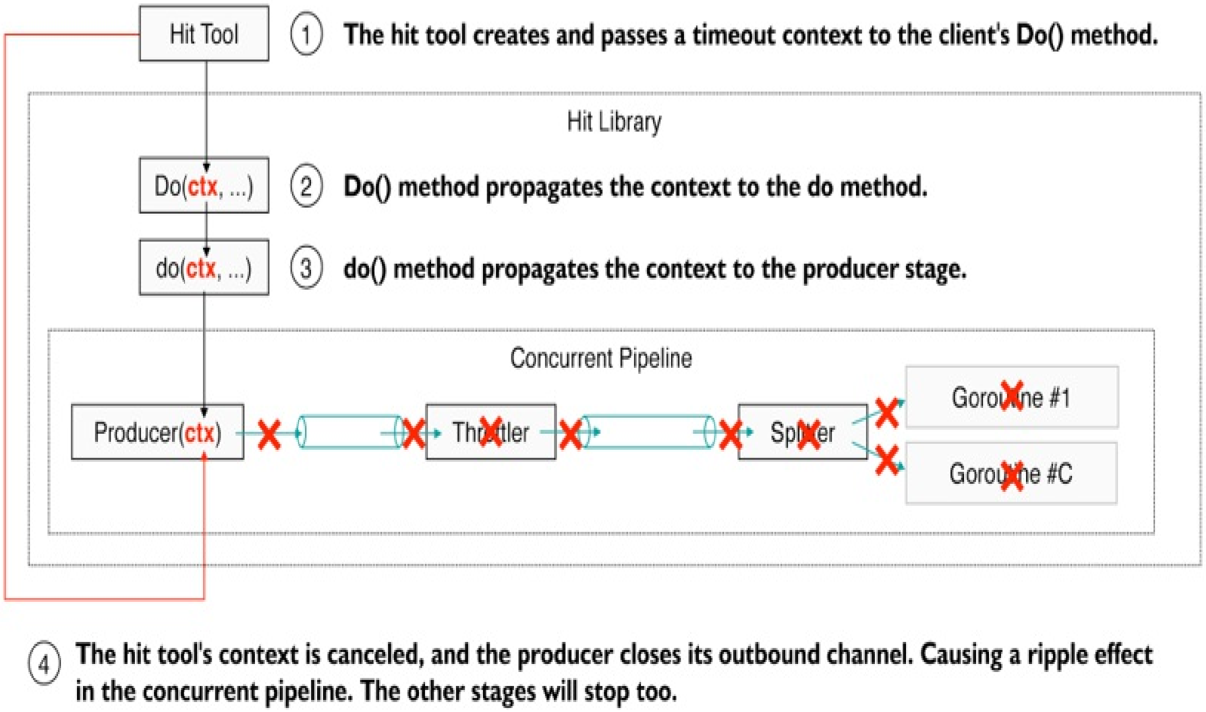
The hit library is unstoppable: Once started, it works until it sends all the requests. How would you stop it? Fortunately, there is a way in the stdlib. The `context` package. This section will teach you how to stop ongoing work using the context package.

6.3.1 The context package: Stopping goroutines gracefully

Spawning goroutines is easy, but shutting them down is not. It's because the Go language does not offer a way to stop a goroutine, at least not directly. Fortunately, the context package provides a straightforward way to stop goroutines. This section will show you how to modify the hit library to control when to stop sending requests.

The hit tool uses the client type's `Do` method to send concurrent HTTP requests (Figure 6.10). If you could pass a context to the `Do` method, you could stop the ongoing requests. The `Do` method can propagate the context to the inner functions and allow them to cancel their work too when the time comes (*when the context is canceled*).

Figure 6.10 The hit tool creates and passes a timeout context to the client. The hit library will stop sending requests when the context is canceled.



If you *stop* the producer, you can *also stop* sending more requests, as other stages in the concurrent pipeline listen to the producer stage. Next, let's modify the producer to accept a context value. Then you'll create a context in the hit tool and pass it to the hit library.

Stopping the producer

Listing 6.12 stops the producer when the context is canceled. The `select` statement listens to both channels and picks the ready one (lines 3-7). The producer keeps sending the generated request values to the outbound channel for other stages to consume.

When the context is canceled, the select statement will pick the context's `Done` channel and cause the producer to close the outbound channel to create a chain reaction in the pipeline to stop the other stages.

Tip

A function that accepts a context should always check if the context is canceled.

Listing 6.12: Stopping the producer (./hit/pipe.go)

```
...
import (
    ...
    "context"
    ...
)
...
func Produce(ctx context.Context, out chan<- *http.Request, n int
    for ; n > 0; n-- {
        select {    #C
        case <-ctx.Done():    #D
            return    #D
        case out <- fn():    #E
        }    #C
    }
}

func produce(ctx context.Context, n int, fn func() *http.Request)
    ...
    go func() {
        defer close(out)    #D
        Produce(ctx, out, n, fn)    #B
    }()
    ...
}
```

Tip

Both channels can become ready at the same time. Since the select statement randomly picks ready channels, it may select the second channel, and the producer can produce one more request value. Eventually, the select will pick the context's channel, and the producer will return. If you want certainty, you could check whether the context is canceled in the function `fn`.

You modified the producer stage and let the producer functions accept a context value. You can now control when to stop the producer.

Notice that the producer functions don't know which context value the caller (e.g., the hit tool) will pass to them (lines 1 and 11). That gives the caller more options for stopping the producer: The caller can pass a context that timeouts after a specific duration. Or, the caller can cancel the outgoing work for other reasons.

Canceling in-flight requests

When the context is canceled, you stopped producing more request values, but what about the ones already in progress? The splitter may have already received the request values from the producer (*or the throttler*) and started sending requests to a server. Could you stop these in-flight requests too?

Fortunately, the http package allows you to stop an in-flight request. You can do it by cloning a request with a cancelable context. Listing 6.13 uses the context while cloning a request value (line 9).

The http package will stop the in-flight requests when the context is canceled. The splitter stage will stop, too, after all the in-flight requests and the producer are stopped. The producer will close its channel after the context is canceled and let the splitter stage know there will be no more requests.

Listing 6.13: Canceling in-flight requests (./hit/client.go)

```
...
import (
    ...
    "context"
    ...
)
...
func (c *Client) Do(ctx context.Context, r *http.Request, n int)
    ...
    sum := c.do(ctx, r, n)
    ...
}

func (c *Client) do(ctx context.Context, r *http.Request, n int)
    p := produce(ctx, n, func() *http.Request {    #B
        return r.Clone(ctx)    #C
    })
    ...
}
```

The `Client` type's `Do` method and its friend, the unexported `do` method, now take a context and propagate it to downstream functions. You also pass the same context while *cloning* a new request value to stop the in-flight requests. Since the `Send` function doesn't send a request to an HTTP server yet, you'll see this in action in the next section (the http package).

Passing a context to the hit library

You made the hit library to be capable of canceling ongoing requests. Well done. It's time to decide when to cancel the ongoing requests.

At the beginning of the section, I told you that you might want to stop requests after a time, let's say after ten minutes. In this subsection, you'll set a shorter timeout to see the effects of cancelation easily. Since the hit tool starts the hit library, it's a great place to create a context.

Let's create a timeout context and pass it to the hit library in Listing 6.14. The `Background` function creates a non-cancelable context (line 5). Then, you use it to derive a *cancelable timeout context* using the `WithTimeout` function (line 5). The timeout context will cancel itself after a second or when you call the `cancel` function. Then you propagate the timeout context to the hit library, and the concurrent pipeline will stop after a second.

It would be more informative to display a user-friendly error message to users. You can do it by checking the context's `Err` method and returning a custom error (lines 10-12). The `Err` method returns `nil` if the context is not yet canceled. However, it returns a `DeadlineExceeded` error if the context is timed-out (line 10). If you were to return the `DeadlineExceeded` error directly, the users would see the following error: "`context deadline exceeded`". It wouldn't be user-friendly, so you customize it with another error (line 11).

Another error you could get is the context canceled error (`Canceled`), which happens if the context is canceled for a reason other than a timeout. You could also catch it and customize it if you want.

Listing 6.14: Creating and passing a context (./cmd/hit/hit.go)

```
...
import (
    ...
    "context"
    ...
)
...
func run(s *flag.FlagSet, args []string, out io.Writer) error {
    ...
    const timeout = time.Second
    ctx, cancel := context.WithTimeout(context.Background(), time
    defer cancel()    #B
    ...
    sum := c.Do(ctx, request, f.n)    #C
    sum.Fprint(out)

    if err := ctx.Err(); errors.Is(err, context.DeadlineExceeded)
        return fmt.Errorf("timed out in %s", timeout)    #D
    }    #D
    return nil
}
```

Note

You can learn more about the errors package's Is function at the link: https://go.dev/blog/go1.13-errors.

You created a timeout context and propagated it to the `Client` type's `Do` method. The `Do` method itself will propagate it to the downstream functions. The ongoing requests will stop when the context is canceled (*after a second*). Lastly, you customized the timeout error message of the context with a user-friendly error message.

It's time to run the hit tool:

```
$ go run . -n 1000 -c 10 http://localhost:9090
        Requests   : 10
        Duration   : 1s
error occurred: timed out in 1s
```

Even though you wanted to send one thousand requests, the hit library sent ten requests. It happened because the context timed out after one second. Celebration time!

6.3.2 The signal package: Is CTRL+C the end?

Imagine you run the hit tool, and after some time, you want to stop it by hitting the CTRL+C keys. Try it! It will stop the hit tool, but you won't get any summary. That is a problem because you might want to see the summary after interrupting the tool.

Hitting CTRL+C will create an interruption signal, and your operating system will deliver it to your program. And your program will terminate abruptly if you're not catching the signal.

Fortunately, you can use the signal package's `NotifyContext` function to catch the signal. Listing 6.15 derives a notification context from the timeout context you created earlier (line 11). The new context will cancel itself when it catches the signal. You call the `stop` function to cancel catching the signals and release the context (*since it creates an internal goroutine)*.

Listing 6.15: Catching interruptions (./cmd/hit/hit.go)

```
...
import (
    ...
    "os/signal"
    ...
)
...
func run(s *flag.FlagSet, args []string, out io.Writer) error {
    ...
    ctx, cancel := context.WithTimeout(context.Background(), time
    ctx, stop := signal.NotifyContext(ctx, os.Interrupt)    #A
    defer cancel()    #B
    defer stop() #C
    ...
}
```

You derived a new notification context from the previous timeout context. In this case, the timeout context is the parent of the notification context. The notification context will be canceled too when you cancel the parent context. However, if you could increase the timeout (*maybe to one minute*), you could hit the CTRL+C keys before the timeout happens.

In that case, only the notification context will be canceled, but not the timeout context since a child context cannot cancel (*and should not!*) its parent context. Luckily, since you call the `cancel` function at the end of the `run` function (thanks to the `defer` statement!), the timeout context will be canceled too and release its acquired resources. It's time to try running the program. Let's run it as follows and hit the CTRL+C keys after some time.

```
$ go run . -n 1000 -c 10 http://localhost:9090
^C
Summary:
        Requests   : 42
        ...
```

Even though I hit the CTRL+C keys after a while, the hit tool showed the summary. A better user experience. Great!

6.3.3 Wrap up

You started the section with a client that never stops. Then you integrated the context package in the client to let it know when to stop sending requests. Let's summarize:

- The `context` package lets you cancel ongoing work.
- The `WithTimeout` function returns a cancellable context that cancels itself after a specified timeout. Or you can cancel it when you call the returned `cancel` function.
- The context's `Err` method returns why the context is canceled. The method returns `DeadlineExceeded` if the cancellation reason is timing out, `Canceled` if the context is canceled for another reason, or `nil` if the context is not yet canceled.
- The `signal` package lets you cancel a context after receiving an interruption signal. The `NotifyContext` function returns a context. The context is automatically canceled after receiving a specified interruption signal from the operating system.

6.4 Sending HTTP requests

You were faking to send HTTP requests to a server. Previously, in Listing

6.4, you wrote a function called `Send` and slept in it to fake sending a request. Let's learn more about the http package's client type and send HTTP requests.

Note

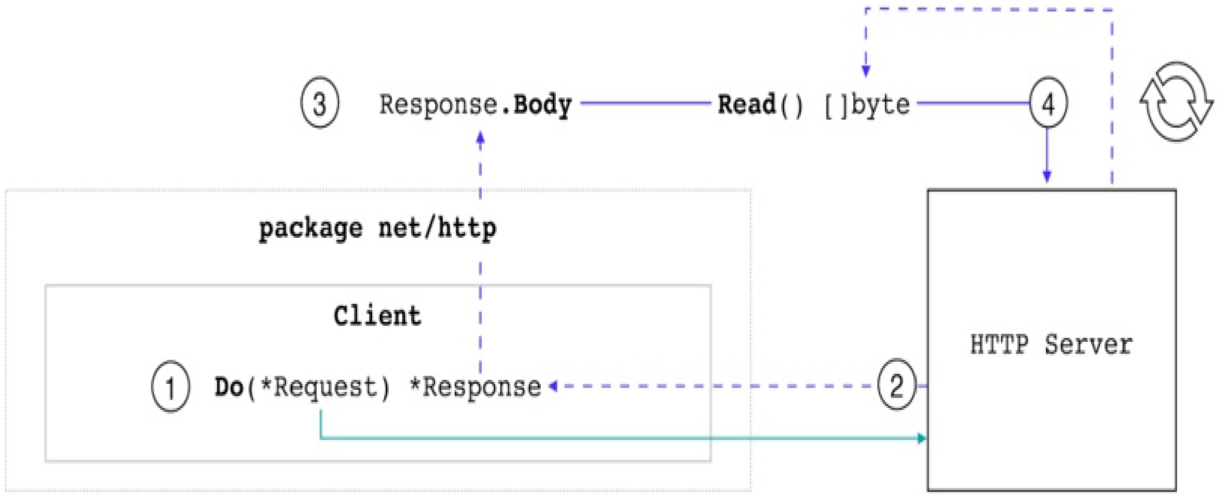
You can learn about the http package at the link https://pkg.go.dev/net/http.

6.4.1 Demystifying the http package's Client type

Let's discuss sending a request to an HTTP server and getting a response. Think of it like requesting your friend to give you a book, and your friend tells you whether he has the book or not (or where) (*HTTP status code*) and gives you the book page by page (*streaming*) instead of handing you the whole book (*response body*). That means you must ask your friend to give you the next page. You can decide what to do with each page, read it and throw it away or store it somewhere. You will need storage space (*computer memory*) if you decide to store the pages.

The http package works similarly (Figure 6.11). You send a request to an HTTP server, and the server responds with an HTTP status code and a response body. You keep asking the server until you read the whole response body. Instead of saving the response body into memory, you read and use a small portion of it.

Figure 6.11 Sending a request and reading the response using the http package.



In Figure 6.11, the `Do` method sends a request using a `Request` value (*step 1*) and gets a response as a `Response` value (*step 2*). The `Response` type represents a response from an HTTP request and contains details such as *status code* and *body* in a field called `Body`. You can use the `Body` field to stream the server's response (*step 3*).

You also want to learn how many bytes are downloaded. In Go, we use the `io` package's `Reader` type when reading a *stream of bytes* from any resource. I say "*any*" because the `Reader` type is an interface, and the `Body` field implements the `Reader` interface. In step 4, you get a response, and you keep streaming the response body as byte slices using the `Read` method until there is nothing left to read or an error occurs.

Note

You can learn more about the `Reader` type at the link https://pkg.go.dev/io/#Reader.

Since the response body is a stream, you need to figure out a straightforward way to read from it. You will get the response body's length but won't save it (*such as to a file or memory*) to reduce memory usage.

Tip

In Go, we use the io package's `Writer` type when writing a stream of bytes to any resource (such as a file or memory). As the `Reader` type, the `Writer` type is also an interface. You can learn more about the `Writer` type at the link https://pkg.go.dev/io/#Writer.

There is a function called `Copy` in the io package to do that. The `Copy` function takes a `Writer` and a `Reader` to write what it reads from the `Reader` to the `Writer`. Then it returns how many bytes are written. Since you want to reduce the memory usage, you can discard what you read using the io package's `Discard` variable. Think of the `Discard` variable as `/dev/null`.

Tip

`Discard` implements the `Writer` interface, and instead of saving data somewhere, it throws it away. You can learn more about it at the link https://pkg.go.dev/io#Discard. Also, you can learn more about the Copy method at the link: https://pkg.go.dev/io#Copy.

6.4.2 Sending an HTTP request

You now know that you can send a request to an HTTP server using the `Client` type's `Do` method and stream the response body using the `Read` method. Instead of using the `Read` method in a loop (*a low-level method*), let's use the `Copy` function to read the response body and the `Discard` variable (a `Writer`) to discard what you read.

In Listing 6.16, you use the http package's default client and send a request (line 8). You stream the response body using the io package's `Copy` method and throw away what you read using the `Discard` variable (line 11). Finally, you close the response body so the http package can *reuse the same connection to the server* (line 12). Since there wouldn't be a body to read if there is an error, you neither need to read the body nor close it (line 9).

Listing 6.16: Sending an HTTP request (./hit/request.go)

```
func Send(r *http.Request) *Result {
    t := time.Now()

    var (
```

```
        code  int
        bytes int64
    )
    response, err := http.DefaultClient.Do(r)    #A
    if err == nil {    #B
        code = response.StatusCode    #C
        bytes, err = io.Copy(io.Discard, response.Body)    #D
        _ = response.Body.Close()    #E
    }    #B

    return &Result{
        Duration: time.Since(t),
        Bytes:    bytes,
        Status:   code,
        Error:    err,
    }
}
```

The `Send` function can now send a request to an HTTP server, learn how many bytes are downloaded, and throw the data away to save memory. You might want to try running the hit tool at this stage with your favorite web server to test it! For example, you can use a test server such as httpbin.org/ as follows to send one thousand requests using ten goroutines:

```
$ go run . -n 1000 -c 10 http://httpbin.org/get
```

Since you created a cancelable context and passed it to the `Send` function in Section 6.3, you can interrupt the previous command in the middle and still see the summary.

6.4.3 Optimizing the HTTP client

Establishing a TCP connection to a server is expensive since doing so requires many back and forth between a client and the server. It's similar to the following joke (each message is sent over the network):

```
Server: Hello, would you like to hear a TCP joke?
  Client: Yes, I'd like to hear a TCP joke.
  Server: OK, I'll tell you a TCP joke.
  Client: OK, I'll hear a TCP joke.
  Server: Are you ready to hear a TCP joke?
  Client: Yes, I am ready to hear a TCP joke.
  Server: OK, I'm about to send the TCP joke. It will last 10 sec
```

```
Client: OK, I'm ready to hear the TCP joke that will last 10 se
<<HTTP request happens here>>
```

You can start sending an HTTP request to the server after establishing a TCP connection. Since establishing a connection is expensive, HTTP protocol has a caching mechanism called *keep-alive*. The server and client can *keep previously established connections open* until the connections *time out*, and a client can use the same connections to send HTTP requests without establishing new ones.

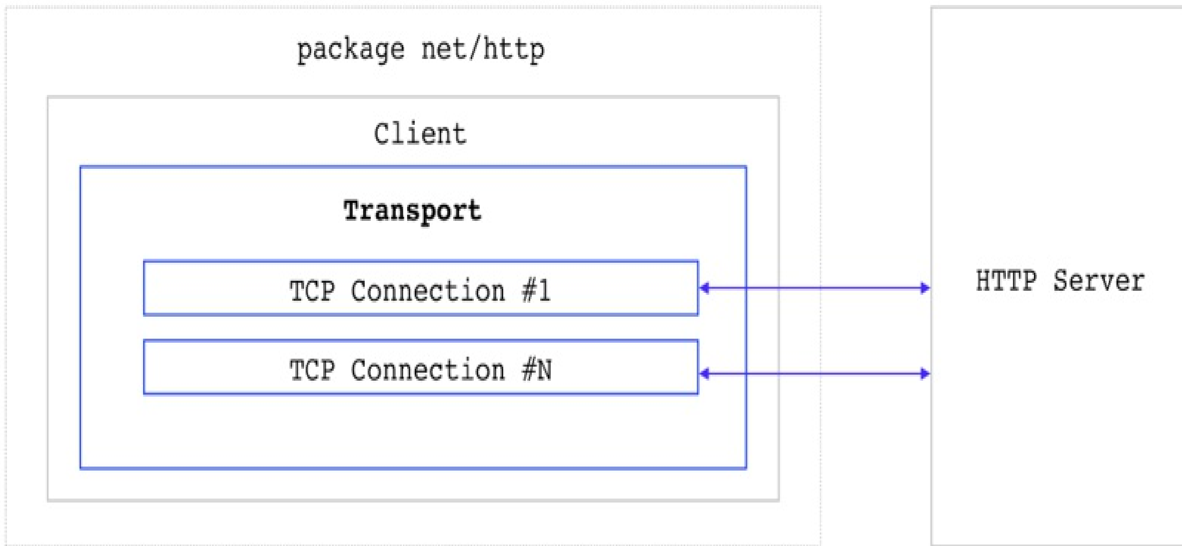
However, the Go Standard Library's default HTTP client configuration (`http.DefaultClient`) keeps one hundred connections open and only allows you to reuse two of them to send an HTTP request to the same host (let's say the same URL)! You should establish a new connection each time to send more than two requests simultaneously.

Let's say the hit library sends thousands of requests using ten goroutines to the same URL. Its performance will be subpar since it uses the `DefaultClient` (The `Send` function in Listing 6.16). While two goroutines can send HTTP requests over two previously established TCP connections, the others would need to open new ones to send more requests. Clearly, the client won't work at full speed yet. This section will show you how to optimize it.

Tweaking the connection pool option

The http package's `Client` type provides a nice API and allows you to send HTTP requests, handles cookies, and redirects. However, it's not the one that establishes TCP connections and sends HTTP requests; it's the `Transport` type (Figure 6.12). The transport establishes TCP connections, sends HTTP requests, and has a connection pool to store *idle* connections. Fortunately, you can configure both!

Figure 6.12 The Go Standard Library's http package's Client and Transport types. The Client uses the Transport to send HTTP requests. The Transport establishes TCP connections, sends HTTP requests, and has a pool for idle connections.



Beware

Since the client (via the transport) has a connection pool, it makes sense to use the same client while making requests. Otherwise, the performance will suffer since there will be multiple clients (hence multiple pools).

Let's start by configuring and returning a new custom `Client` with a custom `Transport` to set the `MaxIdleConnsPerHost` option (lines 17-23). The option matches the concurrency level option of the hit library's client (`C`) so that each goroutine will have a chance to use an idle connection while sending an HTTP request.

Note

If the `MaxIdleConnsPerHost` option were lower than the concurrency level, the transport would frequently close the connections after a request, open new connections for almost every request, and the performance would suffer.

Since you want to take advantage of the connection pooling, the `send` method returns a closure to share the same client for each HTTP request (lines 12-14). Let's also change the `Send` function to accept a customized HTTP client in Listing 6.18.

Recall

In Section 6.1.2, the `SendFunc` type is declared as: `func(*http.Request) *Result`.

Listing 6.17: Customizing Client configuration (./hit/client.go)

```
func (c *Client) do(ctx context.Context, r *http.Request) *Result
    ...
    var (
        sum Result
        client = c.client() #A
    )
    for result := range split(p, c.C, c.send(client)) { ... }
    ...
}

func (c *Client) send(client *http.Client) SendFunc {    #B
    return func(r *http.Request) *Result {    #B
        return Send(client, r)    #A
    }    #B
}

func (c *Client) client() *http.Client {
    return &http.Client{    #C
        Transport: &http.Transport{    #C
            MaxIdleConnsPerHost: c.C,    #D
        },
    }    #C
}
```

Listing 6.18: Accepting a custom client (./hit/request.go)

```
func Send(c *http.Client, r *http.Request) *Result {    #A
    ...
    response, err := c.Do(r)    #A
    ...
}
# Takes a custom HTTP client and uses it to send an HTTP request.
```

Here's what happens when you run the code before you make the customized HTTP client changes. I'm running a local server to see the performance difference (you can use any server URL you like—just make sure that both commands below use the same URL). The following uses the default HTTP client:

```
$ go run ./cmd/hit -n 100_000 -c 10 http://localhost:9090
    RPS         : 2200
    Duration    : 46s
```

And, here's what happens when you use the customized HTTP client:

```
$ go run ./cmd/hit -n 100_000 -c 10 http://localhost:9090
    RPS         : 22000
    Duration    : 4.5s
```

Since the first one didn't effectively use the connection pool, the connection congestion prevented the goroutines from being performant. On the other hand, the second one allowed goroutines to be faster by providing them with reestablished connections from the pool. The performance difference is as clear as day.

With these changes, the hit library's client became more performant. It uses the same HTTP client to take advantage of the HTTP client's idle connection pool while sending HTTP requests. The HTTP client can put a connection to the pool after a request. Then another request can take the connection from the pool to send a request.

Closing idle connections

Imagine you wrote a web server, imported the hit library, and provided an interface to allow others to send HTTP requests and analyze performance results. Your web app is popular, and many people simultaneously send millions of requests to various servers. However, the idle connections in the pool will stay and grow.

After you get the performance result, you could safely throw away the idle connections instead of waiting for them to be closed. Listing 6.19 forcefully closes the idle connections by using the HTTP client's `CloseIdleConnections` method after finishing sending requests.

Listing 6.19: Forcefully closing idle connections (./hit/client.go)

```
func (c *Client) do(ctx context.Context, r *http.Request) *Result
    ...
    var (
        sum Result
        client = c.client()
    )
    defer client.CloseIdleConnections()    #A
    for result := range split(p, c.C, c.send(client)) { ... }
    ...
}
```

With this change, after sending all the requests and getting the performance result, the hit library will close the idle connections in the HTTP client's pool. You can check how many connections are established and closed before and after this change using a network monitoring tool such as `netstat` (https://en.wikipedia.org/wiki/Netstat).

Timing out requests

Imagine a goroutine takes a request value from the pipeline and makes a request, which takes a *long* time. That goroutine wouldn't be able to pick the next request value from the pipeline until it returns a performance result. If all the goroutines have the same problem, the hit library will *never* return, or it will take a *long* time!

If you could define a timeout per request, goroutines could report an error when a specified duration is surpassed instead of clogging the pipeline. Listing 6.20 adds a new `Timeout` field to the `Client` type and sets the HTTP client's `Timeout` option using the new field.

Listing 6.20: Setting a timeout per request (./hit/client.go)

```
type Client struct {
    ...
    Timeout time.Duration // Timeout per request
}

func (c *Client) client() *http.Client {
    return &http.Client{
       Timeout: c.Timeout, // zero means no timeout
       ...
```

```
    }
}
```

The hit library's importers can now set a timeout per request using the new field as follows:

```
c := &hit.Client{
    ...
    Timeout: 10 * time.Second,
}
```

When the importers don't set a timeout, the HTTP client's `Timeout` field will be zero, meaning the request won't timeout. You could set a sensible default if the importers don't set the hit library's `Timeout` field to make the library more convenient to use. I'll leave this out as an exercise for my dear readers.

The HTTP client has many fields you can configure to change its behavior depending on your needs. You can see them at the following links https://pkg.go.dev/net/http#Client and https://pkg.go.dev/net/http#Transport.

6.4.4 Wrap up

You learned about the http package in detail and started sending HTTP requests. Then you optimized the http package's client to send the requests 10x faster! Well done. Let's summarize:

- The `http` package's `Client` type sends HTTP requests. The `Do` method takes a `Request` value and returns a `Response` value.
- The `Response` type represents a response from an HTTP request. The `Response` type's `Body` field implements the `Reader` interface.
- The `io` package's `Copy` function reads from a `Reader` to write to a `Writer`. The `io` package's `Discard` variable is a `Writer` that throws away data.
- The `Transport` type establishes TCP connections, sends HTTP requests, and pools connections. The `MaxIdleConnsPerHost` field controls the number of idle connections to keep in the pool per host. The `Client` type's `CloseIdleConnections` closes the idle connections in the pool.
- The `Client` type's `Timeout` field controls the timeout threshold per request.

6.5 Testing

Imagine you added a new feature to the hit library, and the library does not work anymore. You could easily figure out the problem if you had proper tests in place.

Since the book discussed testing to a great extent, the section's goal won't be testing the library from every angle. If you've read this far, you already know how to test most of the library's code.

This section will focus on *integration testing*. You will learn how to use the `httptest` package to launch a test server in your *testing code* and verify whether the hit library successfully makes a specified number of requests to the test server.

6.5.1 Learning about the HTTP test server

This section will teach you to use the httptest package to launch a test server and handle incoming requests. The test server leaves the decision of how to handle a request to you. That means you can create specific handlers depending on what you want to test.

Note

The httptest package's documentation is https://pkg.go.dev/net/http/httptest.

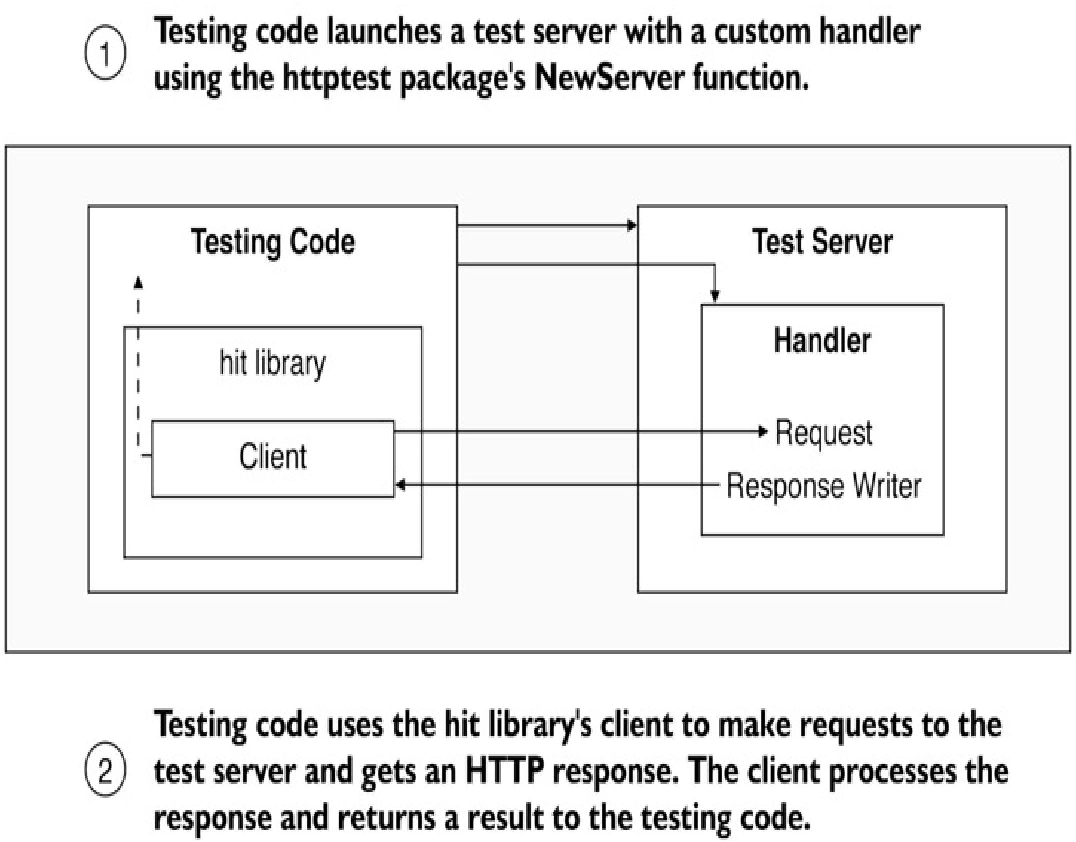
Imagine you want to test successful requests. You can create a handler that responds with HTTP status code 200—*success*. Then you can launch a test server using the httptest package's `NewServer` function with the handler (Figure 6.13). A handler has two ways to communicate with a client: It receives the client's request and returns a response. Think of `Request` as input and `ResponseWriter` as output (I/O).

Note

ResponseWriter's documentation is https://pkg.go.dev/net/http#ResponseWriter.

Finally, you can use the hit library's client to send HTTP requests to the test server and verify whether the client returns a result that says the requests were successful.

Figure 6.13 Testing code launches a test server to test the hit library's client.



Now that you learned that you could launch a test server within testing code using the httptest package's `NewServer` function and a handler, let's discuss what they look like in Figure 6.14.

Note

Handler's documentation is https://pkg.go.dev/net/http#Handler.

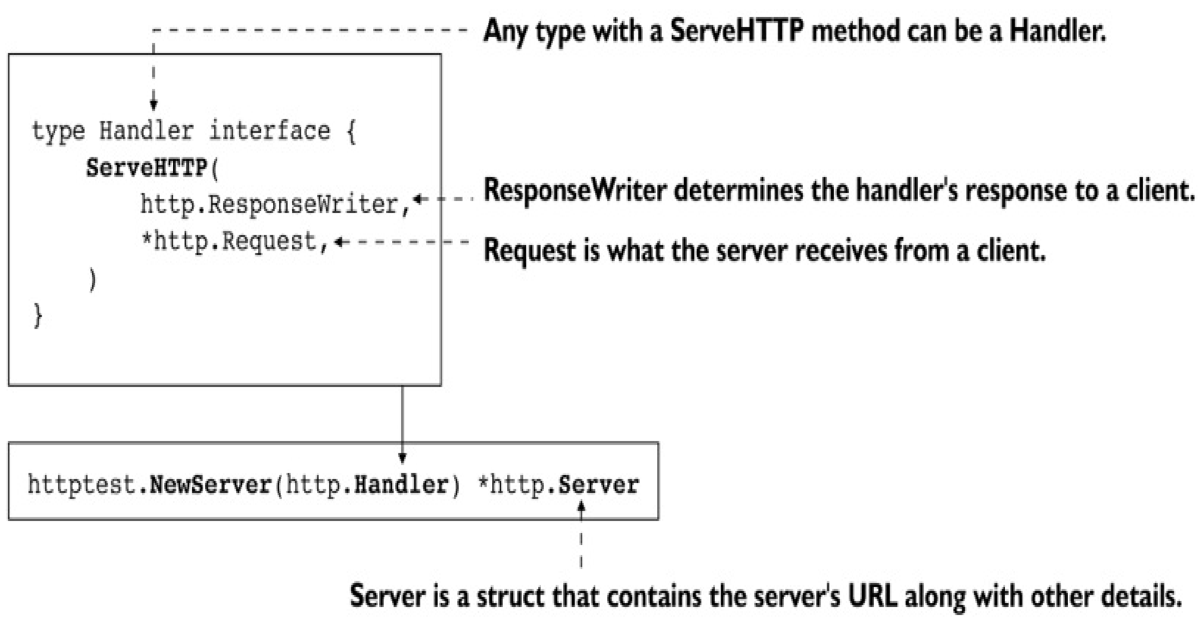
The `Handler` type is an interface, and any type with a `ServeHTTP` method can be a handler. As discussed, a handler gets input from a client as a `Request` value and sends an output to the client using a `ResponseWriter`. You know the `Request` type from the previous sections.

Tip

You can use the `ResponseWriter` to write a response (hence the name!). And you can use the `Request` to learn about the request details (such as the request method (GET, POST, etc.), request URL, remote address, etc.

Once you create a handler, you can pass it to the `NewServer` function to launch a test server. Since the test server would pick a random port to listen to client requests, you would need to learn which port it listens to. Fortunately, the test server returns a `Server` value, and you can use its `URL` field to learn about the server's location.

Figure 6.14 The NewServer accepts a handler and returns a Server value that you can use to find the server's URL. The handler is a way to communicate with a client, and can be any type with a ServeHTTP method.



In summary, a handler can be any type that has a `ServeHTTP` method. The test server accepts incoming requests from a client and packs the request in a `Request` value to pass it to the handler. The test server also passes a `ResponseWriter` to the handler to respond to the client. You can use the test server's URL to send requests.

6.5.2 HandlerFunc: An easy way of using a function as a handler

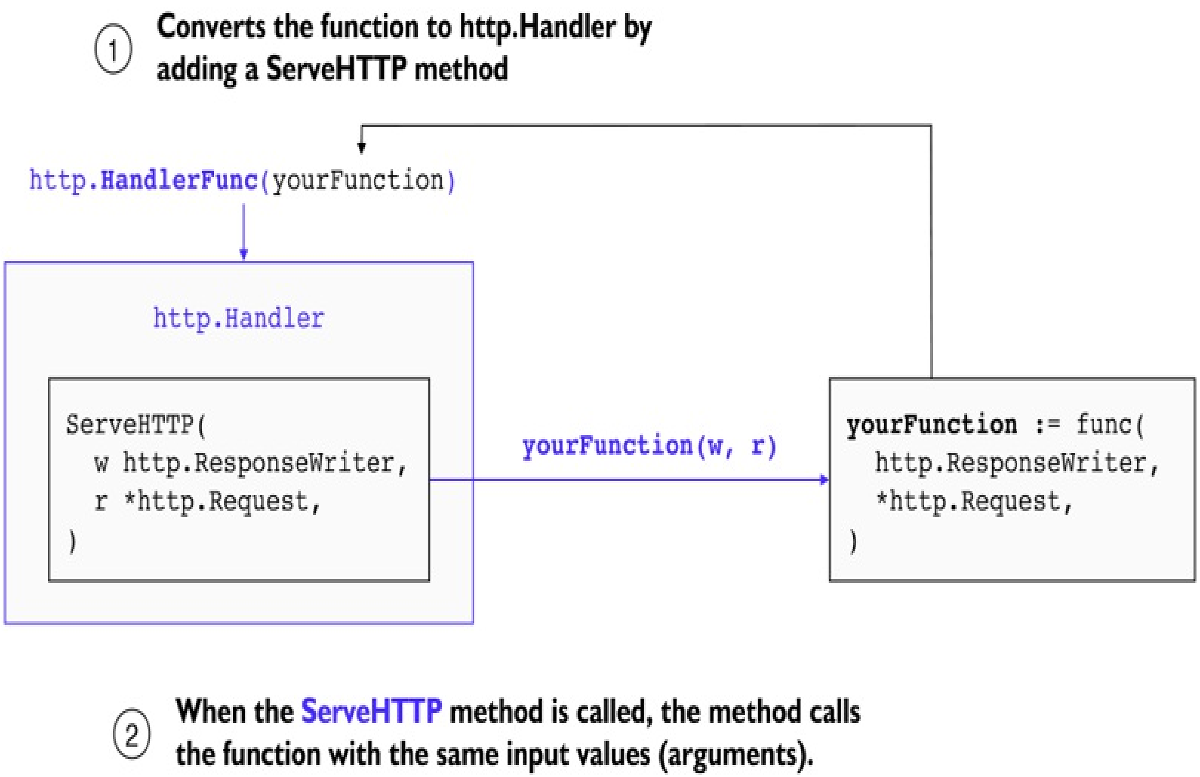
You learned that any type could be a handler if the type has a `ServeHTTP` method. Imagine you want to create a handler that only responds a couple of bytes to the client. The body of such a handler would have a single line of code. Creating a new type and adding a `ServeHTTP` method to make a handler can be tiresome. Fortunately, there is a better way!

Tip

Since a function is a concrete type, *you can add methods to a function*. Moreover, recall from the previous chapter that *you can add methods to a type by converting it to another*. The http package's `HandlerFunc` type uses both techniques to make a function act like a handler. `HandlerFunc`'s documentation is https://pkg.go.dev/net/http#HandlerFunc.

Recall the `ServeHTTP` method from Figure 6.14 and imagine you wrote the method as a function. In Figure 6.15, the `HandlerFunc` type adds the `ServeHTTP` method to the function to convert it to a handler! Since the converted function would have a `ServeHTTP` method, you can pass it to the `NewServer` function to launch a test server. When the test server calls the `ServeHTTP` method, the method will call the underlying function (`yourFunction`).

Figure 6.15 HandlerFunc converts a function to a handler (the figure doesn't show the rest of the function's body for brevity).



The `HandlerFunc` type is an *adapter* (https://en.wikipedia.org/wiki/Adapter_pattern) to convert an ordinary function to a handler as if it had a `ServeHTTP` method. It releases you from declaring a new type to create a handler to the function as a handler to launch a test server.

The method forwards the call to the converted function when the server calls the method:

```
type HandlerFunc func(ResponseWriter, *Request)

func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    // Forwards the call to the converted function.
    f(w, r)
}
```

Since `HandlerFunc` is a function type, you can convert a function with the same signature to the `HandlerFunc` type. Suppose you have a function as follows to handle a request:

```
handler := func(w http.ResponseWriter, r *http.Request) {
```

```
  // handle the request
}
```

Since the function doesn't have a `Serve` method, it does not yet satisfy the `Handler` interface. When you convert the function to the `HandlerFunc` type as follows, the function becomes a `Handler` and satisfies the `Handler` interface:

```
httpHandler := http.HandlerFunc(handler)
// httpHandler now has a Serve method forwarding calls
// to the handler function.
```

Then you can use the handler to launch a test server:

```
server := httptest.NewServer(httpHandler)
```

If you had provided only the handler function, it wouldn't have the `ServeHTTP` method, and you wouldn't be able to pass it to the test server. By converting the function to the `HandlerFunc` type, the function has a `Serve` method that calls the same function!

6.5.3 Testing the client using a test server

Now that you learned what a test server and handler are, let's write a test to verify if the hit library's client can make a specified number of requests to the test server. You will implement a handler to record the number of requests and use it to launch a new test server.

Beware

The test server runs the handler for each request it receives in a separate goroutine.

Since the handler is concurrent, you need to protect the total amount of requests received data to not to be simultaneously changed by the goroutines. The `atomic` package provides concurrency-safe numeric counters which you can use to calculate the total requests received.

Note

Listing 6.21 uses the atomic package of Go 1.18. Please update your Go version if it is older than Go 1.18.

Listing 6.21 declares a concurrency-safe counter called `gotHits` with the atomic package's `Int64` type (line 15)—a concurrency-safe version of the `int64` type. The `Add` method increments the counter whenever the handler receives a request from the client (line 18). The `Load` method returns how many requests the handler has received (line 32).

After declaring the counter, you declare the handler to count the number of requests received (lines 17-19). Since you use the handler solely to increment the counter, you don't use the response writer and request arguments, and skip having them in the handler using *blank-identifiers* (_).

Tip

It's a common practice to use a blank-identifier if you're not planning to use the input value in a function or method.

The time has come to launch the test server to send requests using the hit library's client. You first convert the handler function to a `Handler` using the `HandlerFunc` function and then pass the handler to the `NewServer` function to launch the test server (line 21). You can now get the server's URL to create a new request (line 23). It's critical to *close* the test server after a test run (line 22).

You're finally ready to send requests to the test server. Listing 6.21 creates a client that will send ten requests to the server (lines 28-31). Since you're sending ten requests to the server, the handler should receive an equal amount of requests. The returned result from the client should report the same amount of requests and has no errors.

Listing 6.21: Testing Client.Do (./hit/client_test.go)

```
package hit

import (
    "context"
    "net/http"
```

```
    "net/http/httptest"
    "sync/atomic"
    "testing"
)

func TestClientDo(t *testing.T) {
    t.Parallel()

    const wantHits, wantErrors = 10, 0
    var gotHits atomic.Int64    #A

    handler := func(_ http.ResponseWriter, _ *http.Request) {
        gotHits.Add(1)    #A
    }

    server := httptest.NewServer(http.HandlerFunc(handler))    #B
    defer server.Close()    #C
    request, err := http.NewRequest(http.MethodGet, server.URL, h
    if err != nil {
        t.Fatalf("NewRequest err=%q; want nil", err)
    }

    c := &Client{
        C: 1,
    }
    sum := c.Do(context.Background(), request, wantHits)
    if got := gotHits.Load(); got != wantHits {    #A
        t.Errorf("hits=%d; want %d", got, wantHits)
    }
    if got := sum.Requests; got != wantHits {
        t.Errorf("Requests=%d; want %d", got, wantHits)
    }
    if got := sum.Errors; got != wantErrors {
        t.Errorf("Errors=%d; want %d", got, wantErrors)
    }
}
```

The test launches a test server with a handler to atomically count the number of requests received and tells the testing package to close the server after the test ends. Then the test sends requests to the test server and verifies that the server receives the correct number of requests and has no error.

Let's try it as follows while in the `./hit/` directory:

```
$ go test
PASS
```

Great, it worked! Well done.

6.5.4 Refactoring the test code

The previous test code is fine and readable, but you can polish it using a few reusable helper functions. Listing 6.22 updates the previous code and uses the following helpers:

- The first helper will launch and return a test server.
- The second helper will create and return a new request.

Since the second helper can fail while creating the request value, it will call the testing package's `Fatalf` method to stop the caller test function (in this case, the `TestClientDo` test).

Tip

Test helpers can lead to readable tests and allow reusing the common logic between tests.

The testing package's `Cleanup` function acts like a deferred function and runs a given function after the test ends. The difference between the defer statement and the `Cleanup` function is that you can also use the `Cleanup` function in the test helpers as well.

For example, the `newTestServer` helper will close the test server when the `TestClientDo` test ends. If you were to use a deferred statement in a helper, the helper would call the deferred function at the end of the helper function instead of after the test ends.

Listing 6.22: Refactoring the test (./hit/client_test.go)

```
...
func TestClientDo(t *testing.T) {
    ...
    var (
        gotHits atomic.Int64
        server  = newTestServer(t, func(_ http.ResponseWriter, _
            gotHits.Add(1)
```

```
        })
        request = newRequest(t, http.MethodGet, server.URL)
    )

    c := &Client{
        C: 1,
    }
    sum := c.Do(context.Background(), request, wantHits)
    if got := gotHits.Load(); got != wantHits {
        t.Errorf("hits=%d; want %d", got, wantHits)
    }
    ...
}

func newTestServer(tb testing.TB, h http.HandlerFunc) *httptest.S
    tb.Helper()
    s := httptest.NewServer(h)
    tb.Cleanup(s.Close)
    return s
}

func newRequest(tb testing.TB, method, url string) *http.Request
    tb.Helper()
    r, err := http.NewRequest(method, url, http.NoBody)
    if err != nil {
        tb.Fatalf("newRequest(%q, %q) err=%q; want nil", method,
    }
    return r
}
```

The test creates a new test server with a handler that increments the number of requests received. Then the test creates a new request using the test server's URL. The helpers helped to make the test code more readable and concise.

6.5.5 Providing sensible defaults

Previously you learned how to test the client using the httptest package. However, there is a problem in Listing 6.22. The test will fail when you remove the concurrency field from the client as follows:

```
var c Client
```

You will see the following failure when you run the test:

```
$ go test
--- FAIL: TestClientDo (0.00s)
    hits=0; want 10
    Requests=0; want 10
```

The test unexpectedly failed.

In the last chapter, I suggested having sensible defaults so that the others can comfortably use the packages you created. A sensible default value for the concurrency field could be the number of logical cores on the computer.

Listing 6.23 adds a new method called `concurrency` that returns the concurrency level (lines 20-25). It returns the number of logical cores on the computer if it's not set to have a sensible default. Then it updates the methods that use the concurrency field.

Listing 6.23: Adding the concurrency method (./hit/client.go)

```
...
func (c *Client) do(ctx context.Context, r *http.Request, n int)
    ...
    if c.RPS > 0 {
        p = throttle(ctx, p, time.Second/time.Duration(c.RPS*c.co
    }
    ...
    for result := range split(p, c.concurrency(), c.send(client))
    ...
}
...
func (c *Client) client() *http.Client {
    return &http.Client{
        Timeout: c.Timeout,
        Transport: &http.Transport{
            MaxIdleConnsPerHost: c.concurrency(),
        },
    }
}

func (c *Client) concurrency() int {
   if c.C > 0 {
       return c.C
   }
   return runtime.NumCPU()
}
```

The unexported `concurrency` method does not cache the number of CPUs into a field in `Client` and instead returns it each time it is called. The main reason for not caching it is not to change the `Client`'s fields. To be consistent but unsurprising, users should not see that their `Client` values are mysteriously changed.

Another reason is since `Client` is concurrent, changing the `Client`'s fields could create unexpected data race bugs. You could have decided to cache the concurrency level in the do method, though, and it would be harmless.

Alright, let's try the updated code. You will see the test will pass when you run it:

```
$ go test
PASS
Great!
```

6.5.6 Wrap up

- The httptest package lets you launch a test server within your testing code.
- You can launch a test server using the httptest package's `NewServer` function by giving it a handler that satisfies the `Handler` interface (which has a `Serve` method).
- A handler is a way to handle client requests, and any type with a `ServeHTTP` method can be a handler. A handler receives the client's `Request` and sends a response using the `ResponseWriter` type.
- The `HandlerFunc` type can convert an ordinary function to an HTTP `Handler`.
- The testing package's `Cleanup` method runs a given function after the test ends.
- Provide sensible defaults to users to make your package straightforward to use.

6.6 Refactoring

Although the hit library's code is idiomatic, there is still some work. This section will teach you to refactor the hit library to make it easier to use.

6.6.1 Providing a convenience function

Imagine you want to write a service to periodically check the endurance of one of your services used by millions every day. You started looking for a library package to incorporate into your service and finally found the hit library (what a coincidence).

However, the library needs a few steps that most people don't want to deal with for setting it up. You first need to create a valid request and a client with some options to stress test the service as follows:

```
request, err := http.NewRequest(http.MethodGet, url, http.NoBody)
if err != nil {
    return err
}
var c hit.Client
sum := c.Do(ctx, request, 1_000_000)
// check the website's endurance using the sum variable.
```

Suppose you only want to send one million requests to the service and don't want to deal with the rest of the initialization steps. You could provide a better user experience if the library had a single exported function as follows:

```
sum, err := hit.Do(ctx, url, 1_000_000)
// check the website's endurance using the sum variable.
```

Straightforward to use! The `Do` *function* will be a *wrapper* around the `Client` type's `Do` *method*, making it easy to set up the library to send requests.

In Listing 6.24, the function sends the specified number of requests to a given url using as many goroutines as the number of CPUs on the machine and returns an aggregated result. Behind the scenes, it creates a request—*that's why it returns an error*. Then it creates a new client and calls the `Do` *method*.

Listing 6.24: Implementing the Do function (./hit/client.go)

```
// Do sends n GET requests to the url using as many goroutines as
// number of CPUs on the machine and returns an aggregated result
```

```
//
// Create a new Client to customize sending of requests.
func Do(ctx context.Context, url string, n int) (*Result, error)
    r, err := http.NewRequest(http.MethodGet, url, http.NoBody)
    if err != nil {
        return nil, fmt.Errorf("new http request: %w", err)
    }
    var c Client
    return c.Do(ctx, r, n), nil
}
```

The hit library is now easier to use thanks to the `Do` function. Users can create a `Request` and `Client` themselves if they want to customize the rest of the options.

6.6.2 Refactoring to the self-referential option functions

Previously, you provided a convenience function called `Do` to make it easy to use the hit library. Imagine users want to change the concurrency level but don't want to deal with the hit library's setup steps and keep using the `Do` function. In case they also want to change other options in the future, you can change the `Do` function and add additional input values as follows:

```
func Do(
  ctx context.Context, url string, n, c, rps int, timeout time.Du
) (*Result, error)
```

However, that would complicate the API and confuse the users as they can't see which parameters are optional and would need to provide all. The first three parameters are not optional, but the rest of the parameters are. How can you indicate that some are optional and let the users only provide the options they want?

In his blog, Rob Pike shares an effective pattern called *Self-Referential Option Functions* to solve this problem (also known as the *Functional Options*). The idea is to pass an arbitrary number (*variadic*) of functions as options to change behavior.

Note

You can find more detail about the pattern at the link https://commandcenter.blogspot.com/2014/01/self-referential-functions-and-design.html.

For example, you can pass option *functions* (each takes a `Client` value) to the `Do` function to change the `Client` value's fields. The `Do` function will call each given option function with a client value and let the options change the client value's fields.

In Figure 6.16, you can see what the `Do` function would look like if you had applied this pattern. The function accepts variadic functions to change the client's behavior. The last parameter is *variadic* to allow you to pass zero or more values.

Note

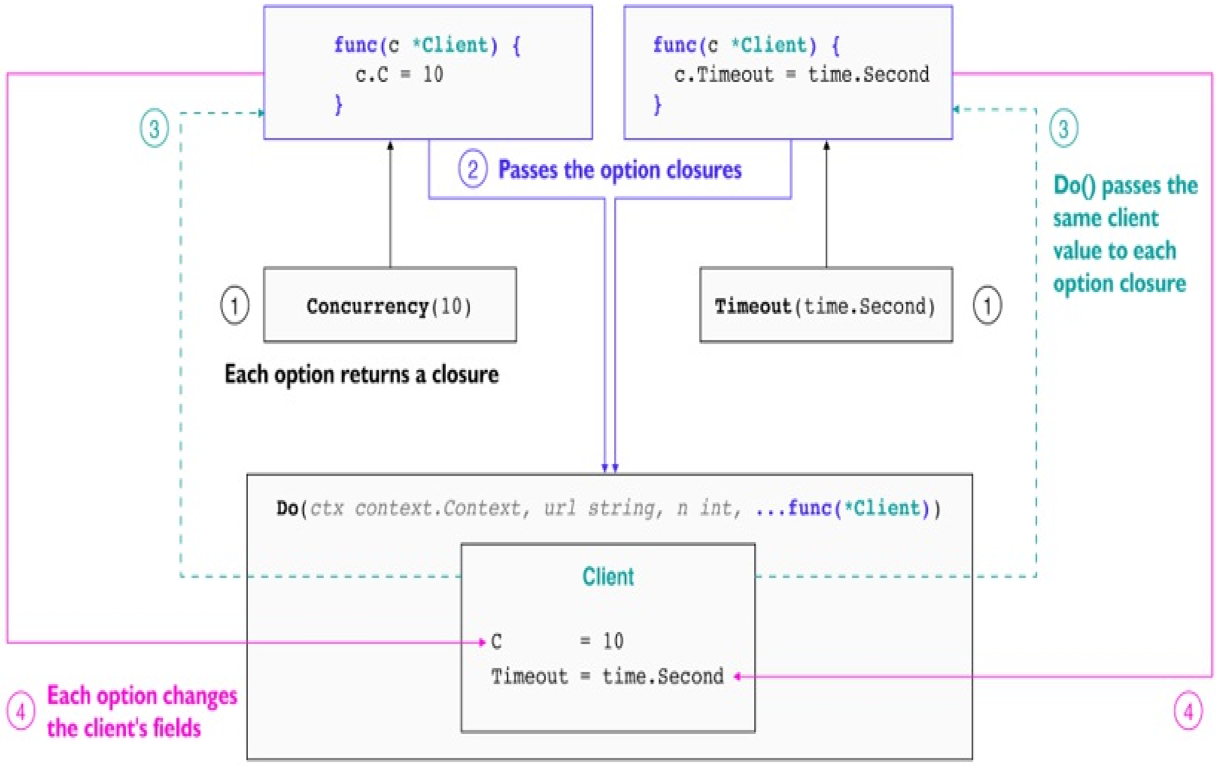
A variadic function accepts an arbitrary number of input values—zero or more. Ellipsis (three-dots) prefix in front of an input type makes a function variadic.

Imagine users only want to change the concurrency level and timeout per request. You can provide two options to let them do that (the first step in Figure 6.16):

- A `Concurrency` option that takes a concurrency level.
- A `Timeout` option that takes a duration value.

As shown in Figure 6.16, each option returns a closure that takes a `Client` value and changes the value. The `Do` function takes variadic options, creates a client, and uses the closures returned by the options to change the client's fields.

Figure 6.16 The Do function takes variadic options, creates a new client value, passes the same client value to each option, and lets them change the client's fields.



1. Each option returns a closure, and each one takes a client value to change.
2. The `Do` function takes the closures and creates a new client value.
3. Since you want the options to configure the same client, the `Do` function passes the same client value to the closures that the options return.
4. Each closure changes one of the fields of the client using the client value. For example, the `Concurrency` option's closure changes the `C` field, and the `Timeout` option's closure changes the `Timeout` field.

In the end, the client's concurrency level becomes ten, and its timeout per request becomes one second.

Now that you understand how the pattern works let's first implement the options and refactor the `Do` function to accept variadic options. Listing 6.25 adds a convenience type called `Option` to keep code concise and more readable. As a side benefit, this type also allows users to see the options in the library's documentation grouped under the `Option` type as follows:

```
type Option
```

```
func Concurrency(n int) Option
func Timeout(d time.Duration) Option
```

Listing 6.25 then implements the options: `Concurrency` and `Timeout`; each returns an `Option` function. For example, the `Concurrency` option takes the concurrency level as an argument, saves it in a closure, and returns the closure that changes the given `Client` value's concurrency level field.

It's time to update the `Do` function and take variadic client options. The `Do` function gets variadic option functions and creates a client. Then, it calls each option function with the same client value to let them configure the client value.

Listing 6.25: Refactoring the Do function (./hit/client.go)

```
...
type Client { ... }

// Option allows changing Client's behavior.
type Option func(*Client)    #A

// Concurrency changes the Client's concurrency level.
func Concurrency(n int) Option {    #B
    return func(c *Client) { c.C = n }    #C
}

// Timeout changes the Client's timeout per request.
func Timeout(d time.Duration) Option {
    return func(c *Client) { c.Timeout = d }
}

func Do(ctx context.Context, url string, n int, opts ...Option) (
    ...
    var c Client
    for _, o := range opts {    #E
        o(&c)    #E
    }    #E
    return c.Do(ctx, r, n), nil
}
...
```

There are two options; each returns a function to change a given client value. Users can now pass the options to the `Do` function to change how the `Do` function will send requests. The `Do` function creates a new `Client` value and changes its fields using the options.

Tip

You could set additional default values for the new client before overwriting the client fields with the option functions. However, I don't suggest doing it there. Instead, you can follow the way you did with the concurrency function in Listing 6.23. Doing so will allow users to get the same default behavior when they create a new `Client` or use the `Do` function.

Now that you implemented the pattern, let's see how you can use it. For example, the following will change the client's concurrency level and timeout per request fields:

```
hit.Do(
    ctx, "http://somewhere", 1_000,
    hit.Concurrency(10), hit.Timeout(time.Second),
)
```

Thanks to the variadic arguments, users don't have to provide every option, and they can decide to change only the concurrency level as follows:

```
hit.Do(
    ctx, "http://somewhere", 1_000,
    hit.Concurrency(10),
)
```

Since the pattern allows an arbitrary number of options to configure the `Client`, users don't have to provide any options at all, preserving the original behavior of the API—*making the API straightforward to use*:

```
hit.Do(ctx, "http://somewhere", 1_000)
```

The `Do` function wraps the underlying `Client` type's `Do` method and makes it easy to send HTTP requests. It allows others to customize sending HTTP requests by accepting variadic option functions. An option function returns a closure to change the target value's behavior indirectly. Other fellow programmers (and maybe you) will thank you when you provide a straightforward-to-use API.

6.6.3 Wrap up

You now have a function called `Do` that can do everything the hit library `Client` can. Others no longer need the `Client` type to send HTTP requests. Maybe it's time to keep the API surface even simpler by unexporting the `Client` type (as `client`) and letting people only use the `Do` function instead!

Then again, some might need a specialized client type. Fortunately, the hit library has every functionality—*composable like LEGO bricks*—they would need to build a custom client:

- The `Result` type to save, merge, and print request results.
- The `Send` function to send an HTTP request and get a `Result`.
- Concurrent pipelining functions: `Produce`, `Throttle`, and `Split`.

I leave unexporting the `Client` type to you as an exercise. You might also want to add a `Request` option to let others customize requests they want to make.

6.7 Exercises

1. Since the throttler (Sections 6.2.3 and 6.2.5) slows down the pipeline, it can take a little while for the producer's turn to come to check if the context is canceled. That can cause a delay in the concurrent pipeline stopping. Immediately stop the throttling stage when the context is canceled.
2. Add a new timeout flag to the hit tool for the hit library client's `Timeout` field.
3. Add a field with the http package's `Client` type to the hit library's `Client` type to let others fine-tune the HTTP client to their needs. Also, implement an option function to pass to the `Do` function.
4. Create an example test (see Chapter 4) to show users how to use the `Do` function.
5. Thoroughly test the hit library's `Client` and target a coverage ratio of at least 80%.

6.8 Summary

Congrats! You now have an idiomatic HTTP client with a straightforward-to-use API. You can use the tricks you learned in this and the previous chapter to build idiomatic command-line tools and libraries.

- Achieving an effective architecture is mostly about reducing complexity by dividing a task into composable parts where each will be responsible for doing a smaller set of tasks.
- API is what you export from a package. Hide complexity behind a simple and synchronous API and let other people decide when to use your API concurrently. Concurrency is an implementation detail.
- Concurrency is structuring a program as independently executing components. A concurrent pipeline is an extensible and efficient design pattern consisting of concurrent stages. You can easily add and remove stages and compose different pipelines without changing stage code.
- Spawning goroutines is easy, but shutting them down is not. It's because the Go language does not offer a way to stop a goroutine, at least not directly. Fortunately, the context package provides a straightforward way to stop goroutines.
- The http package allows you to send HTTP requests, and the httptest package can launch a test server to test code that sends HTTP requests.
- Rob Pike's option functions pattern lets you provide a customizable API without complicating the API surface area.

7 Designing an HTTP Service

This chapter covers

- Structuring, writing, running, securing, and testing an idiomatic HTTP server.
- Using middleware and handler chaining patterns to minimize repetitive code and add extra functionality to HTTP handlers without changing their code.
- Receiving and responding with the JSON format.

You're about to embark on an exciting journey at Linkit, a start-up with ambitions to transform the world of link management. Your first project at Linkit will be a *URL shortener API* which shortens long URLs based on a user-specified short key. When a user requests the shortened URL, the server redirects the user to the long URL. For example, users might use the short key `"go"` to go to `"https://go.dev"`.

This chapter will guide you in building the URL shortener HTTP API using Go's `net/http` package, following your introduction to handlers, requests, and responses in the last chapter.

Note

You can find the source code for this chapter at https://github.com/inancgumus/effective-go/tree/main/ch07.

Package structure

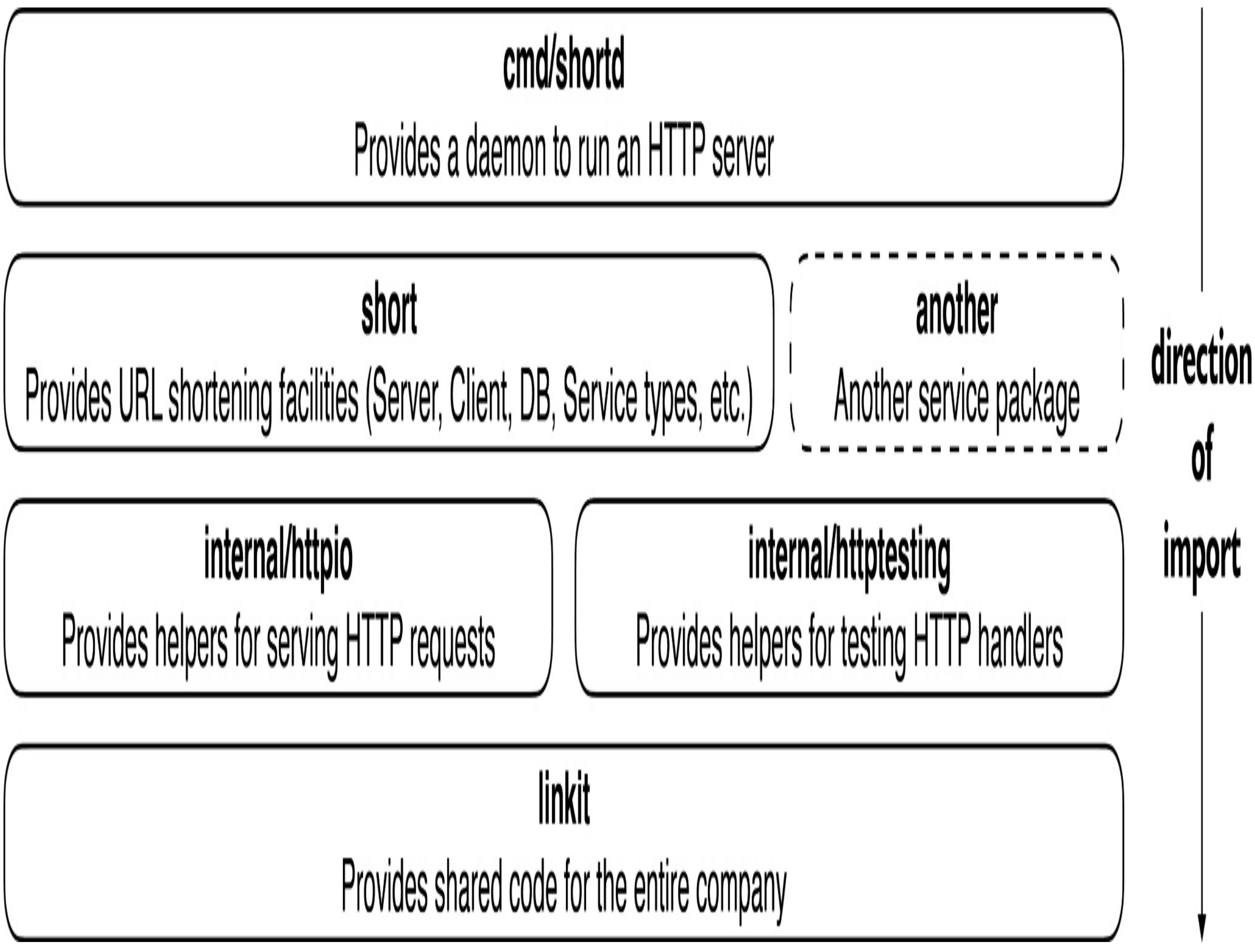
In Go, circular dependency is a compile time error where package A imports package B and package B imports package A.

Aren, a savvy gopher teammate, knows that organized code is key to a project's success. As a way to avoid circular dependencies and keep your code easy to maintain, he suggests the layered package structure shown in

Figure 7.1 for the URL shortener project.

As a way to avoid circular dependencies, let packages import from those below them. For instance, the `short` package can import any package beneath it, but the `httpio` package shouldn't import from the `short` package. As the `short` package expands, you can shift functionality to sub-packages under the `short` folder and only reference them from the `short` package.

Figure 7.1 The layered package structure prevents circular dependencies by allowing higher-level packages to import from lower-level ones. This approach keeps your code organized and provides flexibility when adding new packages.



Additionally, the structure deliberately avoids catch-all packages like `common`, `util`, or `models` that often accumulate unrelated dependencies and become tricky to maintain. Instead, the package names have a clear purpose for a more organized and comprehensible codebase.

Tip

The `internal` folder restricts other packages from importing the `httpio` or `httptesting` packages unless they share the same directory root with the `internal` folder. This keeps the packages private to your project, reducing coupling and enhancing code maintenance. For more information, visit https://go.dev/doc/go1.4#internalpackages.

Lastly, using an `internal` folder is optional. You can simply add an ordinary "`// do not import`" comment within a package to discourage usage outside its intended scope (Go loves pragmatism, after all).

By following the structure in Figure 7.1, you'll keep dependency trees under control, minimize unrelated dependencies and maintain clarity. This structure also simplifies adding new services or executables, and each service can be deployed independently.

There is no single right way

Embrace the freedom of not being tied to a specific architecture, structure, or framework. Start with a single package and add more as needed. Experiment to find what works best for you and your team. Stick to a simple and pragmatic approach, and avoid introducing unnecessary abstractions. Remember, YAGNI (You Aren't Gonna Need It).

7.1 Writing an HTTP server

Your team is tasked with creating the *URL shortener server*. In this section, you'll gain insights into the following aspects to help you achieve that goal:

- The core concepts of the `net/http` package.
- Running and securing an HTTP server.
- Routing incoming requests to handlers for shortening and resolving URLs.

By mastering these fundamentals, you'll be well-equipped to tackle more advanced topics later in the chapter.

7.1.1 Learning about the core concepts

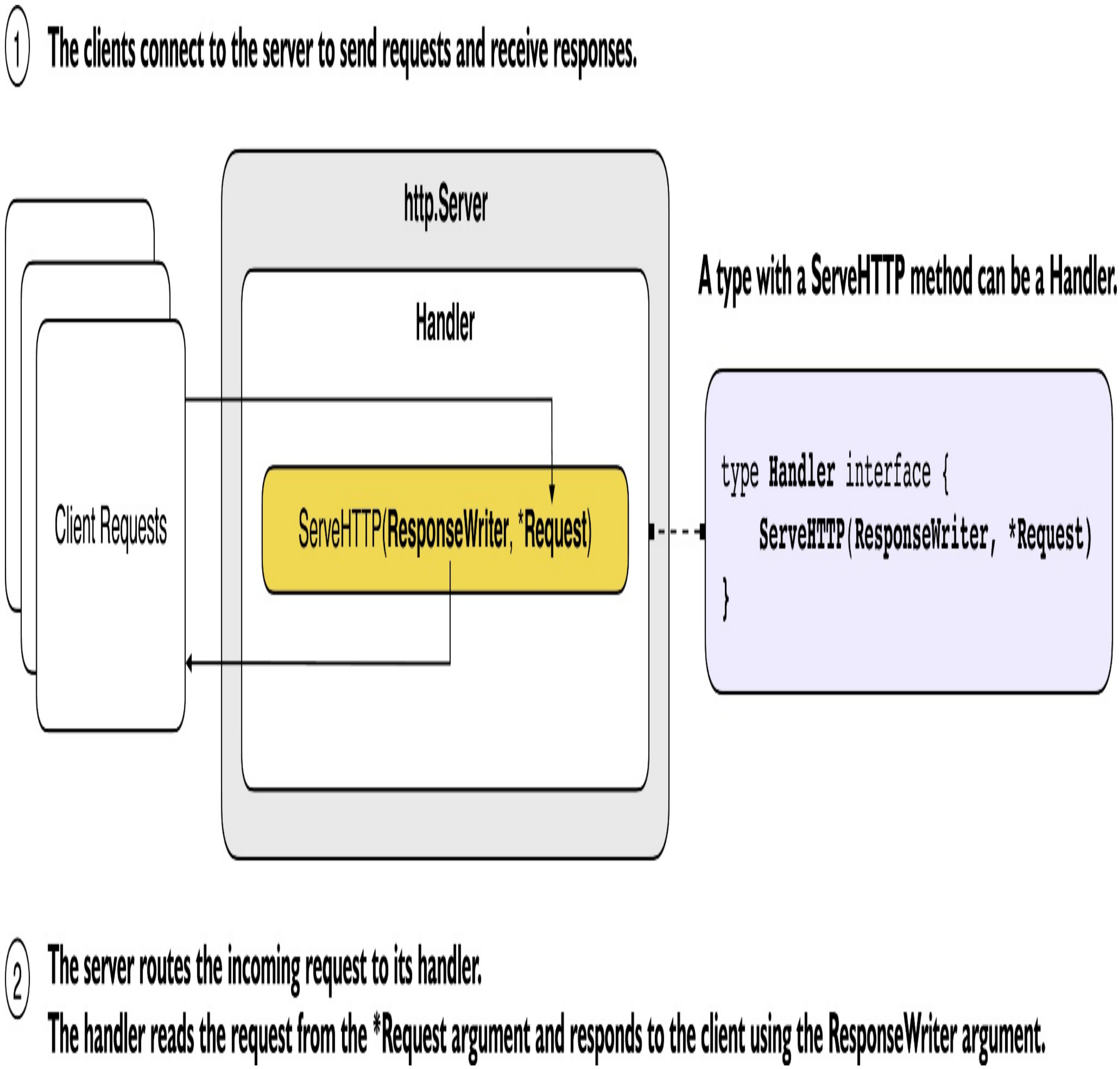
Let's start by exploring the `Server` and `Handler` types in the `net/http` package.

As Figure 7.2 shows:

- The `Server` is a type to listen for client connections and to route incoming requests to a type that satisfies the `Handler` interface.
- The `Handler` interface is elegantly simple and has a single method: `ServeHTTP`.

Whenever a request comes in, the `Server` reads it in a new goroutine and hands it over to a `Handler` by calling its `ServeHTTP` method. Inside the handler, you can use `ResponseWriter` to respond to the client and `*Request` to inspect details about the incoming request. The `Server` provides these values for each request allowing you to focus on the request and response handling logic in your handler.

Figure 7.2 Server routes incoming client requests to a handler.



You can use a type with a `ServeHTTP` method to handle HTTP requests. However, creating a type to satisfy the `Handler` interface is not always practical. Instead, you can convert a regular function (or a method) with the following signature to a `Handler` using the `HandlerFunc` function:

```
myFunc    := func(w http.ResponseWriter, r *http.Request) { /* ..
myHandler := http.HandlerFunc(myFunc)    #B
```

Now that you're familiar with the basics of the `Server` and `Handler` types, let's discover how to set up and launch a server to listen for incoming connections and handle HTTP requests.

Note

For more information about the `net/http` package, visit: https://pkg.go.dev/net/http.

7.1.2 Running the server

Let's dive in and write a program to launch a server to listen for client connections and implement a handler to process incoming requests. You'll use the `http.ListenAndServe` helper function—which looks like the following, to launch a new `http.Server`:

```
// The server listens on addr for client connections.
// The handler serves incoming HTTP requests.
func ListenAndServe(addr string, handler http.Handler) error
```

In Listing 7.1, `ListenAndServe` launches a new `Server` underneath to *listen* for client connections on `localhost:8080` and *serves* incoming HTTP requests with the `shortener` handler.

Note

The `ListenAndServe` function blocks until the server stops and returns a *non-nil* error. Don't worry about the `ErrServerClosed` error; it's an expected error when the server stops without issues, and you can safely ignore it.

For each incoming request, the server calls the `shortener` handler, and the handler writes a message to the client using the `Fprintln` function. The `ResponseWriter` type is a `Writer` with a `Write` method, so you can use `Fprintln` to write to the client.

Listing 7.1: Running the server (./cmd/shortd/shortd.go)

```
package main
// imported packages here

func main() {
    const addr = "localhost:8080"

    fmt.Fprintln(os.Stderr, "starting the server on", addr)
```

```
    shortener := http.HandlerFunc(        #A
        func(w http.ResponseWriter, r *http.Request) {
            fmt.Fprintln(w, "hello from the shortener server!")
            // w.Write([]byte("hello from the shortener server!")
        },
    )
    err := http.ListenAndServe(addr, shortener)    #B
    if !errors.Is(err, http.ErrServerClosed) {
        fmt.Fprintln(os.Stderr, "server closed unexpectedly:", er
    }
}
```

Since the server and handler are ready to handle incoming requests, it's time to test the server by running it and making a request from another terminal using any tool you prefer.

```
$ go run ./cmd/shortd
starting the server on localhost:8080
$ curl -i localhost:8080
HTTP/1.1 200 OK
hello from the shortener server!
```

Note

Remember to stop the server with CTRL+C (or Command+C) and rerun it whenever you make changes to the code; otherwise, the server will run the old code.

7.1.3 Hardening the server

Imagine your HTTP server crashing due to a client request that took too long to respond. To avoid this in the future, you decide to set up timeouts. Since `ListenAndServe` doesn't allow setting timeouts, let's create a new `Server` value yourself to configure timeouts.

In Listing 7.2, the `Addr` field specifies the network address where the server listens for incoming connections. The `ReadTimeout` field sets the maximum duration for reading the entire request from a client. The `Handler` field specifies the handler to handle incoming requests, which, in this case, is `TimeoutHandler` which wraps a handler and returns itself:

```
// TimeoutHandler wraps h and returns a timeout handler.
func TimeoutHandler(h http.Handler, dt time.Duration, msg string)
```

As shown in Listing 7.2, once the server receives a request, `TimeoutHandler` gets the request before the `shortener` handler since it wraps around the `shortener` handler.

- `TimeoutHandler` responds with a timeout error to the client if the `shortener` handler runs for more than ten seconds.
- `TimeoutHandler` also clones the request with a new `Context` and sets a timeout. This allows you to pass the `Context` to a long-running operation in the handler, check if it's been canceled, and then stop it.

Listing 7.2: Setting timeouts (./cmd/shortd/shortd.go)

```
func main() {
    const (
        addr    = "localhost:8080"
        timeout = 10 * time.Second
    )
    fmt.Fprintln(os.Stderr, "starting the server on", addr)

    shortener := http.HandlerFunc(
        func(w http.ResponseWriter, r *http.Request) {
            fmt.Fprintln(w, "hello from the shortener server!")
        },
    )
    server := &http.Server{    #A
        Addr:          addr,
        Handler:       http.TimeoutHandler(shortener, timeout, "ti
        ReadTimeout:   timeout,
    }
    if err := server.ListenAndServe(); !errors.Is(err, http.ErrSe
        fmt.Fprintln(os.Stderr, "server closed unexpectedly:", er
    }
}
```

Setting timeouts will help ensure your server remains operational despite problems with client requests.

Using HTTPS

To protect against man-in-the-middle attacks, you can launch the server using the `ListenAndServeTLS` method (similar to `ListenAndServe`) to listen and respond over HTTPS connections. Visit https://pkg.go.dev/net/http#Server.ListenAndServeTLS to learn more.

7.1.4 Serving with multiple handlers

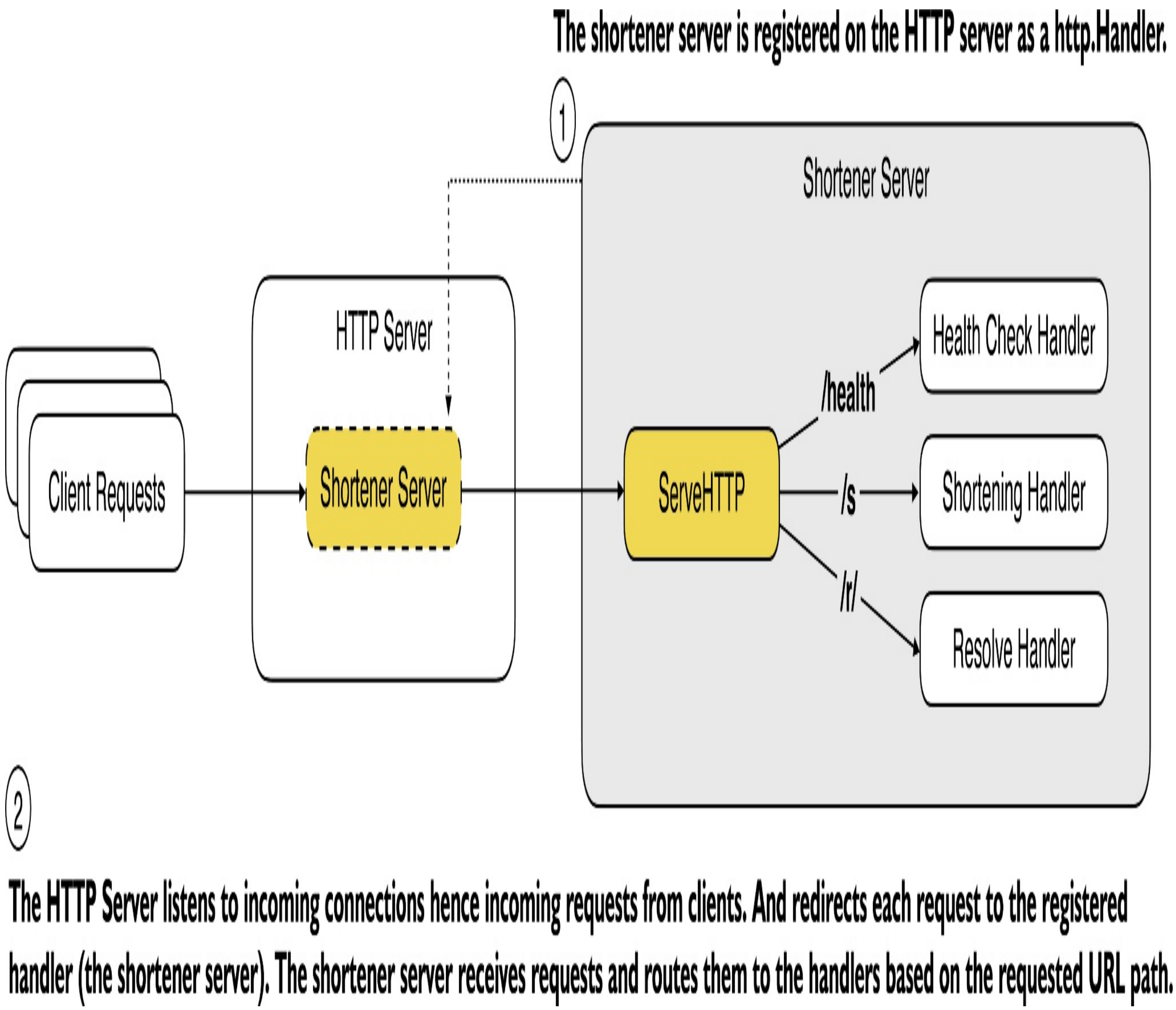
You implemented a server with a single handler and ran it, but you ran into a limitation: the `Server` type *only allows for a single handler*. In a real-world service with multiple handlers, like the URL shortener server, relying on a single handler won't suffice. You need the ability to route incoming requests to the appropriate handlers based on the requested route.

Recall

When a type has a `ServeHTTP` method, it can be registered on the `Server` as a handler. Then the `Server` type can forward incoming requests to the handler.

To solve this issue, consider Figure 7.3, where you see a new *shortener server type* that acts as a router. The new type satisfies the `Handler` interface and is registered on the `http.Server` as a handler, where it routes the requests to the handlers.

Figure 7.3 The HTTP server (http.Server) routes incoming requests to the shortener server, which in turn routes the requests to the appropriate handlers.



Now that you know the URL shortener server will act as *both* a handler and a router, it's time to implement it and route incoming requests. I'll guide you through the implementation process and show you how to integrate it with the daemon you previously created.

Note

You may observe that the `resolveRoute` has a trailing slash (`/`). URL patterns ending with `/` act as prefix patterns, matching any URL path starting with that pattern and additional segments. This helps manage multiple URLs sharing a common prefix. The `resolveRoute` pattern uses `/` to match any path starting with `/r/` followed by a key (`/r/shortkey1`, `/r/shortkey2`, etc.). Conversely, `shorteningRoute` and `healthCheckRoute` patterns lack `/`, as they only match the precise URL path.

Implementing the handlers

Let's start implementing the URL shortener server in Listing 7.3 by declaring a new type called `Server` in the `short` package with the following handlers.

1. The health check handler helps load balancers or orchestrators like Kubernetes determine if the server is ready to handle the traffic.
2. The shortening handler responds with an HTTP status code of `Created` and the text "`go`" when a client requests to shorten a URL.
3. Lastly, the resolve handler redirects client requests to go.dev from the key.

Tip

Putting your handlers as methods in a type can help you achieve maintainability and allow them to access server-specific resources (i.e., database) readily.

To make the URL shortener server itself a `Handler`, you implement the `ServeHTTP` method and route incoming requests to the handlers by extracting the incoming request's route path (such as `/r/`, `/s`, or `/health`) using the `Path` field.

Listing 7.3: Implementing the URL shortener server (./short/server.go)

```
package short
// imported packages here

const (
    shorteningRoute  = "/s"
    resolveRoute     = "/r/"
    healthCheckRoute = "/health"
)

type Server struct {}

func NewServer() *Server { return &Server{} }
```

```
func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request
    switch p := r.URL.Path; {
    case p == healthCheckRoute:
        s.healthCheckHandler(w, r)
    case strings.HasPrefix(p, resolveRoute):
        s.resolveHandler(w, r)
    case strings.HasPrefix(p, shorteningRoute):
        s.shorteningHandler(w, r)
    default:
        http.NotFound(w, r) // respond with 404 if no path matche
    }
}

func (s *Server) healthCheckHandler(w http.ResponseWriter, r *htt
    fmt.Fprintln(w, "OK")    #B
}

func (s *Server) shorteningHandler(w http.ResponseWriter, r *http
    w.WriteHeader(http.StatusCreated)    #C
    fmt.Fprintln(w, "go")    #B
}

func (s *Server) resolveHandler(w http.ResponseWriter, r *http.Re
    const uri = "https://go.dev"
    http.Redirect(w, r, uri, http.StatusFound)    #D
}
```

Although it may not be the most advanced routing mechanism (`ServeHTTP`), it provides a solid foundation. Even though the handlers have hard-coded responses, having the basic structure in place is an excellent starting point for building out the full functionality.

Tip

Prefix or suffix a handler function with `Handle` or `Handler` to differentiate them from non-handler functions or methods. Your text editor and future self will thank you.

By default, handlers write a status code of OK

The health check handler, in particular, won't write a status header using `WriteHeader`, but it will still respond with an HTTP status code of `OK`. Handlers will automatically do that if you don't specify a different status code using the `WriteHeader` method before calling `Write`.

Integrating with http.Server

Nice job on creating the URL shortener server, incorporating a router, and setting up the handlers! Now it's time to connect the server with the daemon. All you need to do is assign a new URL shortener server to the `shortener` variable.

Listing 7.4: Integrating with the server (./cmd/shortd/shortd.go)

```
...
func main() {
    ...
    shortener := short.NewServer()
    server := &http.Server{
        ...
        Handler: http.TimeoutHandler(shortener, timeout, "timeout
        ...
    }
    ...
}
```

The `Handler` field requires a type that satisfies the `Handler` interface (a type with a `ServeHTTP` method). Since the URL shortener server has a `ServeHTTP` method, you can assign it to the `Handler` field.

With the implementation of the URL shortener server and its router, incoming requests can now be effectively directed to the appropriate handlers based on route patterns. It's time to put it to the test and see it in action!

```
$ curl -i localhost:8080/health
HTTP/1.1 200 OK
OK
$ curl -i localhost:8080/s
HTTP/1.1 201 Created
go
$ curl -i localhost:8080/r/go
HTTP/1.1 302 Found
Location: https://go.dev
```

Multiplexing with ServeMux

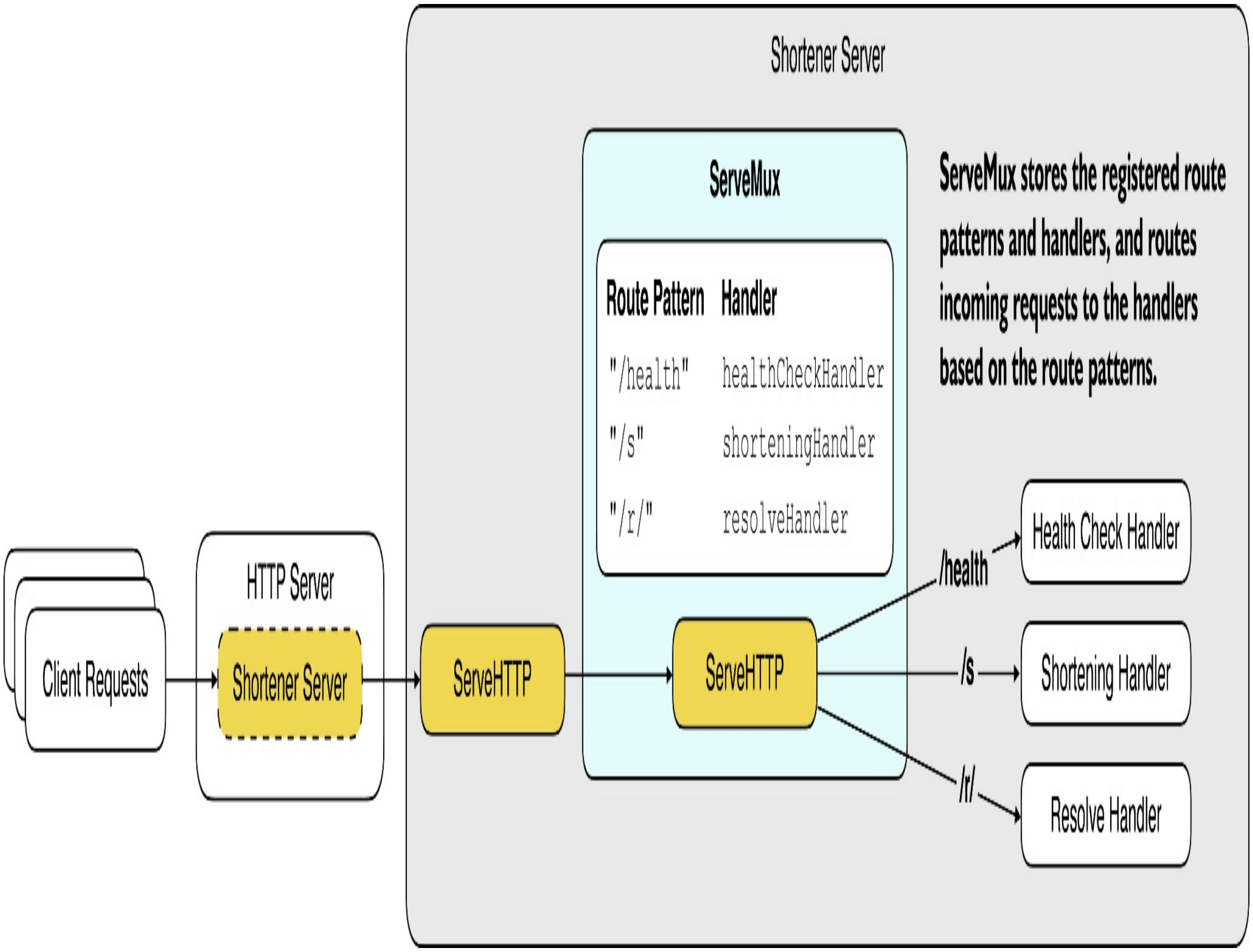
The router you implemented in the previous section can become a hassle when you want to add more routes. To avoid scaling issues when manually routing requests to handlers, you can use `ServeMux`—also called a *mux* or *muxer*—to register your handlers and take care of the routing.

Note

`ServeMux` is a `Handler` and has a `ServeHTTP` method.

When a request arrives, the `ServeHTTP` method of the shortener server is invoked, which, in turn, calls `ServeMux.ServeHTTP` to forward the request to a matching handler that corresponds to the registered route pattern.

Figure 7.4 The URL shortener server will register its handlers on a new `ServeMux` and delegate the routing responsibility to `ServeMux`.



Now that you know `ServeMux` can route incoming requests to handlers, let's modify the previous router and incorporate `ServeMux` into the shortener server in Listing 7.5, where it creates a new `ServeMux` to register the handlers and automate the routing. By converting each handler method into a `Handler` using `HandleFunc` and calling the `ServeHTTP` method of `ServeMux`, you can delegate routing responsibilities to `ServeMux`.

Listing 7.5: Using ServeMux (./short/server.go)

```
type Server struct {
    mux *http.ServeMux    #A
}

func NewServer() *Server {
    var s Server
    s.registerRoutes()
    return &s
}
```

```
func (s *Server) registerRoutes() {
    mux := http.NewServeMux()    #A
    mux.HandleFunc(shorteningRoute, s.shorteningHandler)    #B
    mux.HandleFunc(resolveRoute, s.resolveHandler)    #B
    mux.HandleFunc(healthCheckRoute, s.healthCheckHandler)    #B
    s.mux = mux    #A
}

func (s *Server) ServeHTTP(w http.ResponseWriter, r *http.Request
    s.mux.ServeHTTP(w, r)    #A
}
```

You've added a new `mux` field and allowed it to route incoming requests to the shortener server handlers. Now, let me share a tip to make this code even more streamlined and maintainable.

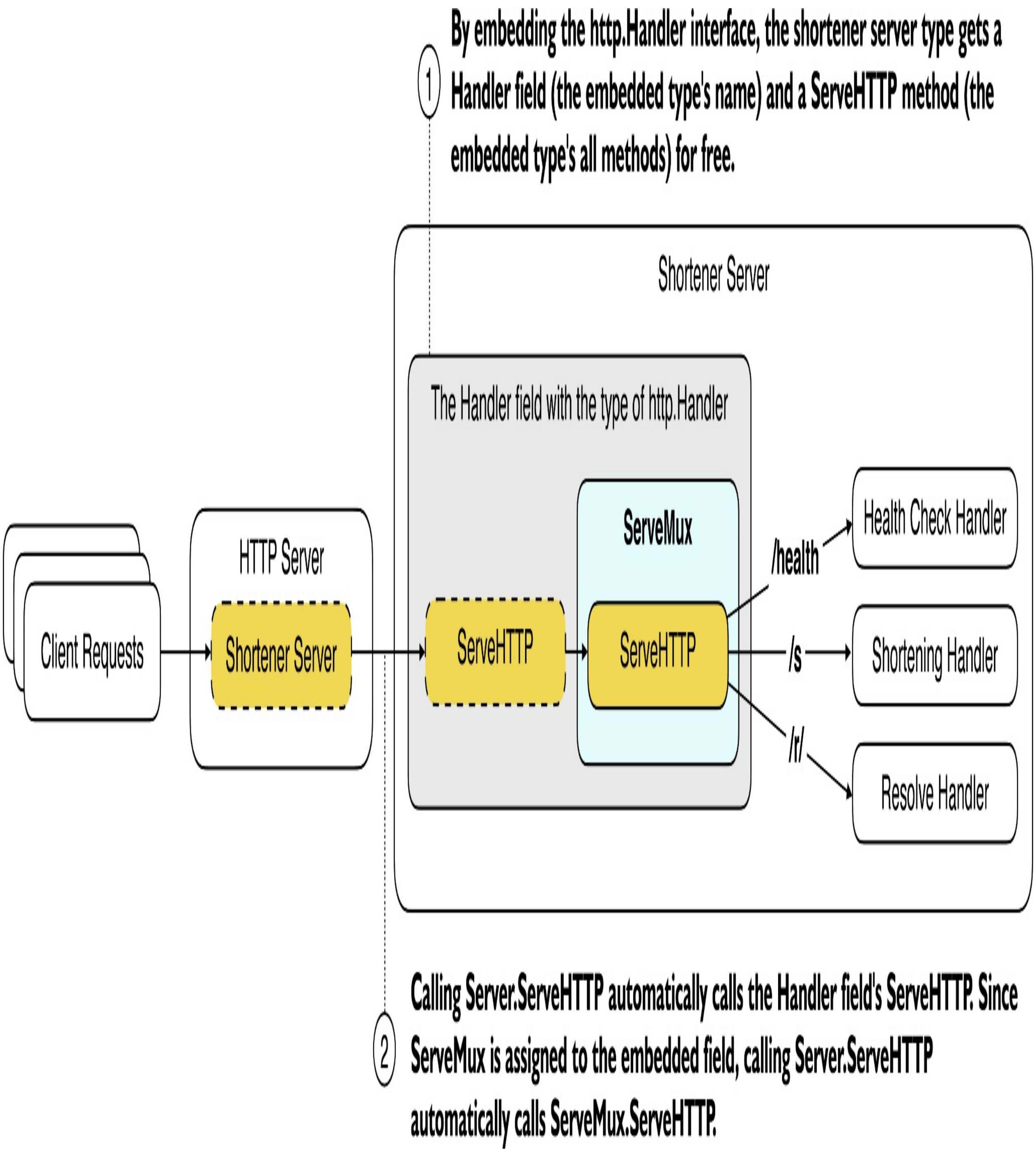
Note

Visit the documentation for more details: https://pkg.go.dev/net/http#ServeMux.

Interface embedding

As shown in Figure 7.5, here's a neat little trick to avoid repetitive code and keep things clean and organized: You can *embed* the `Handler` interface in the URL shortener server and let the embedding type (`Server`) to have the embedded type's (`Handler`) methods (`ServeHTTP`) for free. Since the `Handler` interface has only a `ServeHTTP` method, the `Server` will have a `ServeHTTP` method. Once you assign `ServeMux` to the `Handler` field, `ServeMux` can route all incoming requests to the registered handlers (since `ServeMux` is a `Handler`).

Figure 7.5 Embedding a type allows you to have the embedded type's methods as if they were the methods of the embedding type. Embedding an interface lets you change behavior in the compile-time and runtime.



To embed a type, you simply specify its type name without a field name. When you embed a type, the field name is automatically set to the embedded type's name. For instance, in Listing 7.6, the `http.Handler` type is embedded into the `Server` type and assigned a `Handler` field, which allows you to assign the muxer (`ServeMux`) to the field.

Listing 7.6: Using ServeMux (./short/server.go)

```
type Server struct {
    http.Handler    #A
}

func (s *Server) registerRoutes() {
    mux := http.NewServeMux()
    ...
    s.Handler = mux
}
```

Beware

Remember to delete the `ServeHTTP` method from the `Server` type.

It's important to note that embedding is not inheritance and there are still two values: The URL shortener `Server` and `ServeMux`—promoting *composition over inheritance*. Method calls on the embedding type are merely *delegated* to the embedded type, maintaining their distinct identities. Visit the link for more information: https://go.dev/doc/effective_go#embedding.

Hiding the embedded type

While embedding the `Handler` interface allowed for easy routing of incoming requests to handlers, it had a flaw: the embedded field is exported, allowing users to assign a different handler from outside the `short` package. Listing 7.7 addresses the issue by creating an unexported type (`mux`) and then embedding this new type.

Listing 7.7: Hiding the router (./short/server.go)

```
type mux http.Handler

type Server struct {
    mux // Server will only export ServeHTTP
}

func (s *Server) registerRoutes() {
    mux := http.NewServeMux()
    ...
```

```
    s.mux = mux
}
```

Since you embedded the `mux` type, `Server` still includes a `ServeHTTP` method for free. With all your routes now registered on a new `ServeMux`, you can direct incoming requests to `ServeMux`, eliminating the need for a custom router. And with that, you can call it a day!

Limit what you export from a type to reduce coupling

Embedding the new `mux` type instead of the concrete `ServeMux` allows the URL shortener server to gain an additional `ServeHTTP` method (without exposing the `ServeMux`'s other methods) while enabling the assignment of any type that satisfies the `Handler` interface, including third-party routers.

7.1.5 Wrap up

In this section, you've gained valuable skills for creating and managing HTTP servers, such as processing requests, directing them to various handlers, customizing server settings, and incorporating timeouts to guard against rogue clients. These abilities are crucial when building sturdy and dependable web applications.

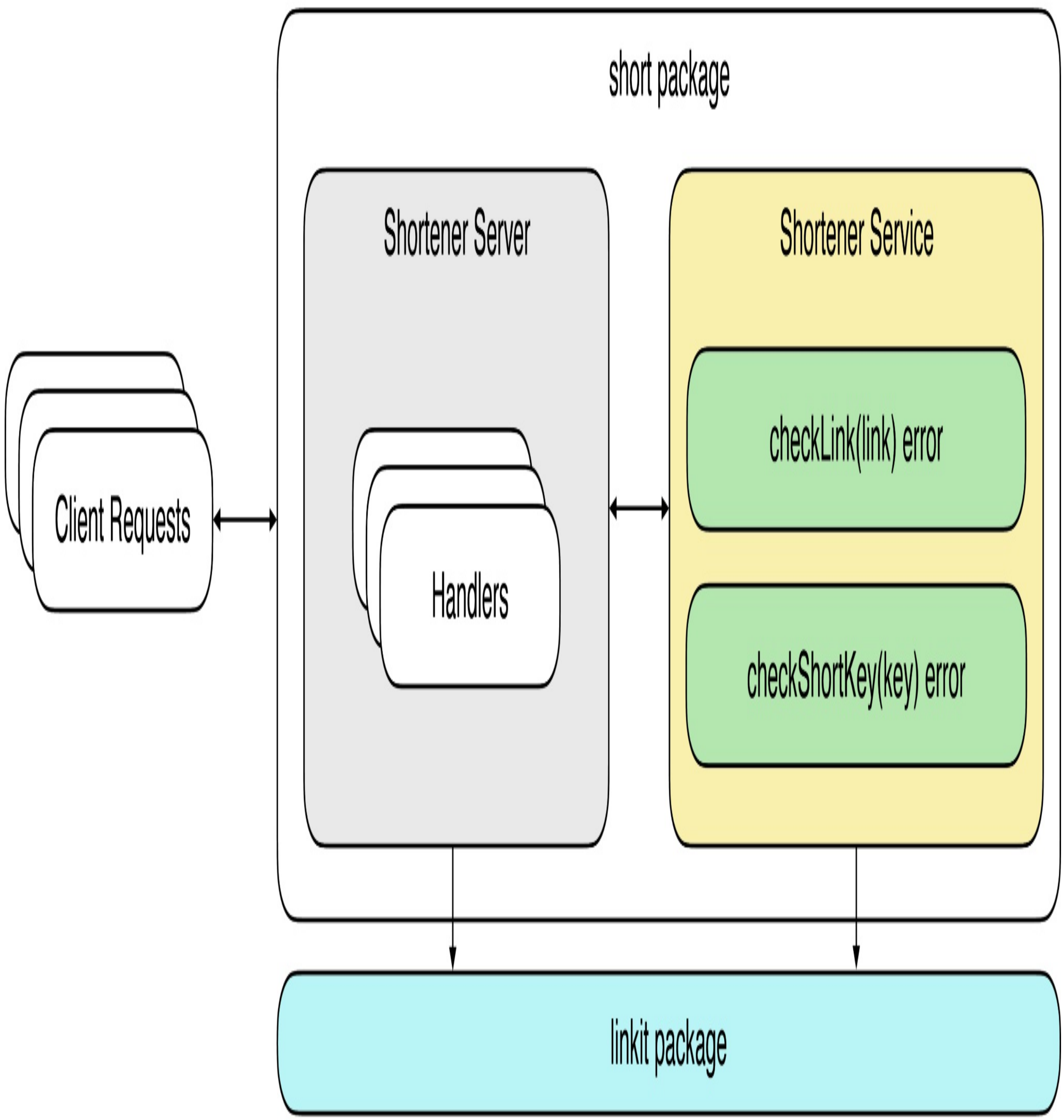
Moreover, interface embedding boosts flexibility and minimizes redundancy by letting you use methods from an embedded type as though they belonged to the embedding type. This technique also facilitates seamless integration with third-party routers compatible with the `Handler` interface, further expanding your options.

7.2 Implementing the service

As shown in Figure 7.6, the service logic acts like a busy kitchen in a restaurant, taking care of essential tasks like validating requests, processing data, and returning errors based on business rules. The HTTP handlers grab the requests, pass them to the service for processing, and then send the response back to the client, just like a helpful server delivering an order.

Figure 7.6 The service and handlers collaborate to handle client requests and generate responses in a coordinated manner.

The service provides domain-specific types, rules, and does validation.



Provides common functionality such as errors.

Your teammate Lisa thinks that integrating service logic in the handlers is a smart move since it verifies the data sent to the handlers and makes your system more reliable.

7.2.1 Service logic

Lisa wants to ensure that all parts of the code speak the same language, making the codebase consistent and easy to maintain. To achieve this, she creates a new package containing shared errors that can be imported and used in handlers and future services.

Listing 7.8: Adding common errors (./linkit/error.go)

```
package linkit

import "errors"

var (
    ErrExists   = errors.New("already exists")
    ErrNotExist = errors.New("does not exist")
    ErrInternal = errors.New("internal error: please try again la
)
// other shared types—structs, interfaces, etc., may be in differ
```

You'll often come across these standard errors in the service and other parts of the code. You can add more types to the `linkit` package as needed, such as a user management service. Anything that should be shared can be included in this package.

Defining the business rules

Lisa believes that clear data structures are essential for a consistent codebase. She wants to create the `link` type to define the data the service deals with. This well-defined data structure makes writing efficient and reliable code much easier for the whole team.

Listing 7.9: Adding the link type (./short/short.go)

```
package short
```

```
const maxKeyLen = 16

type link struct {
    uri      string
    shortKey string
}
```

Using structs is helpful because it allows your team to add new fields to the structs in the future without breaking most of your code. This ensures that the data will always have a consistent structure, making code more readable and easier to understand. Next, she adds service functions to make sure clients can only send proper data to the service.

Listing 7.10: Adding the service rules (./short/short.go)

```
func checkLink(ln link) error {
    if err := checkShortKey(ln.shortKey); err != nil {
        return err
    }
    u, err := url.ParseRequestURI(ln.uri)    #A
    if err != nil {
        return err
    }
    if u.Host == "" {
        return errors.New("empty host")
    }
    if u.Scheme != "http" && u.Scheme != "https" {
        return errors.New("scheme must be http or https")
    }
    return nil
}

func checkShortKey(k string) error {
    if strings.TrimSpace(k) == "" {
        return errors.New("empty key")
    }
    if len(k) > maxKeyLen {
        return fmt.Errorf("key too long (max %d)", maxKeyLen)
    }
    return nil
}
```

To reduce coupling between packages and make the code easier to maintain, avoid exporting the service functions since they are only used internally by the `short` package. Think carefully before exporting your functions and other identifiers.

7.2.2 Making handlers smarter

Now that Lisa has provided a solid business core let's use the service logic she has developed in the handlers in Listing 7.11.

Note

`Request` type's `Method` field returns the request's method (GET/POST/etc.).

Start by getting the request from the client, then delegate the request processing to the service. Check for any errors in the handlers; if there are any, respond with friendly errors that the clients can understand.

Listing 7.11: Applying the service logic (./short/server.go)

```
func (s *Server) shorteningHandler(w http.ResponseWriter, r *http
    if r.Method != http.MethodPost {    #A
        http.Error(w, "method not allowed", http.StatusMethodNotA
        return
    }
    ln := link{
        uri:      r.FormValue("url"),
        shortKey: r.FormValue("key"),
    }
    if err := checkLink(ln); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return    #B
    }
    w.WriteHeader(http.StatusCreated)
    w.Write([]byte(ln.shortKey))    #C
}

func (s *Server) resolveHandler(w http.ResponseWriter, r *http.Re
    key := r.URL.Path[len(resolveRoute):]

    if err := checkShortKey(key); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return    #B
    }
    // use dummy data for now and carelessly expose internal deta
```

```
    if key == "fortesting" {
        http.Error(w, "db at IP ... failed", http.StatusInternalS
        return
    }
    if key != "go" {
        http.Error(w, linkit.ErrNotExist.Error(), http.StatusNotF
        return    #B
    }
    const uri = "https://go.dev"
    http.Redirect(w, r, uri, http.StatusFound)
}
```

Tip

Remember to return from a handler if you want to stop processing the request.

The shortening handler gets the URL and short key from the request using the `FormValue` method and puts them in a new `link`. The resolve handler gets the short key from the requested `URL`'s `Path` and strips the route path: "`/r/go`" becomes "go". The resolve handler also adds a hard-coded condition to test for internal errors, which will become useful.

Let's give it a try:

```
$ curl -i localhost:8080/s -d 'url=https://go.dev&key=go'
HTTP/1.1 201 Created
go
$ curl -i localhost:8080/r/go
HTTP/1.1 302 Found
Location: https://go.dev
```

Let's make a GET request:

```
$ curl -i localhost:8080/s -XGET -d 'url=https://go.dev&key=go'
HTTP/1.1 405 Method Not Allowed
method not allowed
```

Let's query the server with a non-existing short key:

```
$ curl -i localhost:8080/r/skeleton
HTTP/1.1 404 Not Found
does not exist
```

Currently, the handlers don't do much except work with dummy data. In the next chapter, you'll add storage functionality.

7.2.3 Wrap up

In this section, you learned how service logic and HTTP handlers collaborate to handle requests and responses. You also understood the importance of using consistent data structures, shared errors, and only exporting necessary service functions. This knowledge sets you up for the next section.

7.3 Encoding and Decoding JSON

Imagine the customers need JSON support in the shortener server for seamless software integration. To make it happen, you'll learn how to decode and encode JSON data in the handlers using Go's powerful `json` package.

Note

Visit the documentation for more information: https://pkg.go.dev/encoding/json and the blog post (https://go.dev/blog/json). Look especially for `Marshal` and `Unmarshal`.

Adding helpers to encode and decode JSON

Before diving into adding JSON support for the URL shortener server handlers, let's first check out some handy helpers to make things easier. In Listing 7.12, you'll add a new package called `httpio`, which provides useful functions for dealing with JSON data:

- `Decode` reads from a `Reader` into `v` and doesn't allow unexpected fields.
- `Encode` writes a Go value as JSON to the client.

This versatile package is designed for ease of use and can be expanded with more helpers.

Note

The empty interface (`any` or `interface{}`) is a catch-all interface type that can be used to represent any type. This lets you pass any type to the `v` argument of the `Encode` and `Decode`.

Listing 7.12: Adding JSON helpers (./internal/httpio/httpio.go)

```
package httpio
// imported packages here

func Decode(r io.Reader, v any) error {
    decoder := json.NewDecoder(r)
    decoder.DisallowUnknownFields()
    return decoder.Decode(v)
}

func Encode(w http.ResponseWriter, code int, v any) error {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(code)
    return json.NewEncoder(w).Encode(v)
}
```

With these useful JSON helpers ready, it's time to update the shortening handler to utilize them. You can update the resolve handler later since it's just for redirection.

Tip

To ensure the client receives the correct content type, you should set the header before calling `WriteHeader`, and never call the `Write` method before setting the headers.

Making handlers speak JSON

Suppose the shortening handler receives a JSON request in this format:

```
{"url": "https://go.dev", "key": "go"}
```

To process this request, in Listing 7.13, the handler creates a new type called `input` to define the expected structure of the incoming data. One way to decode the JSON data is by *exporting the fields* as follows.

```
struct {
  URL string
  Key string
}
```

Tip

The `json` package only considers exported fields while encoding and decoding.

When the JSON request reaches the shortening handler, the decoder examines the struct fields, decodes the JSON data, and stores it in the fields of the `input` variable. In this scenario, `input.URL` will contain "`https://go.dev`" and `input.Key` will hold "go".

Then the handler encodes the short key using a map value—where the keys are strings and values can be any type of value, and sends the encoded JSON to the client. For more complex JSON data, you might want to use a struct value instead.

Listing 7.13: Using JSON in handlers (./short/server.go)

```
func (s *Server) shorteningHandler(w http.ResponseWriter, r *http
    // ... http.MethodPost checking code
    var input struct {    #A
        URL string    #B
        Key string    #B
    }
    err := httpio.Decode(r.Body, &input)    #C
    if err != nil {
        http.Error(w, "cannot decode JSON", http.StatusBadRequest
        return
    }
    ln := link{
        uri:      input.URL,
        shortKey: input.Key,
    }
    if err := checkLink(ln); err != nil {
        // ... error handling code
    }
    _ = httpio.Encode(w, http.StatusCreated, map[string]any{
        "key": ln.shortKey,
    })
```

```
}
```

Now, the shortening handler can accept and send responses in JSON format:

```
$ curl -i localhost:8080/s -d '{"url": "https://go.dev", "key": "
HTTP/1.1 201 Created
{"key":"go"}
The shortening handler decodes the client's JSON request by readi
```

Tip

To be more flexible, it is best to avoid mixing input and output types with domain types.

Remember, the `input` variable (an anonymous struct) is only used for decoding JSON data and is separate from the `link` type, representing data within the app. Keeping these types apart allows you to modify the `input` type without affecting the domain type and vice versa. This way, you can load the domain type from a file or database or change it elsewhere in the code without worrying about altering the `input` type.

Hardening the reader

Let's reconsider Listing 7.13. The shortening handler reads the `Request`'s `Body` without any restrictions. Security is vital, so limiting the maximum number of bytes a handler can read from a client request is a great way to safeguard your server.

In Listing 7.14, the `MaxBytesReader` function wraps the `Request`'s `Body`, stopping the reading process after 4,096 bytes and returning an error. This approach helps prevent attacks from malicious clients.

Listing 7.14: Using MaxBytesReader (./short/server.go)

```
func (s *Server) shorteningHandler(w http.ResponseWriter, r *http
    ...
    err := httpio.Decode(http.MaxBytesReader(w, r.Body, 4_096), &
    if err != nil {
        http.Error(w, "cannot decode JSON", http.StatusBadRequest
        return
```

```
    }
    ...
}
```

#A `Decode` reads from `MaxBytesReader`, which in turn reads from `Request.Body`.

By using `MaxBytesReader`, the server gains an extra layer of security, shielding it from malicious clients that might send large amounts of data. This enhancement makes the server more robust and secure.

Wrap up

The `json` package's `Decoder` and `Encoder` types simplify working with JSON data in HTTP handlers by enabling the decoding of JSON data into Go values and the encoding of Go values into JSON data. The `MaxBytesReader` function provides extra security by wrapping the `Request`'s `Body` and limiting the reading to a specified maximum number of bytes, protecting the server from potential issues.

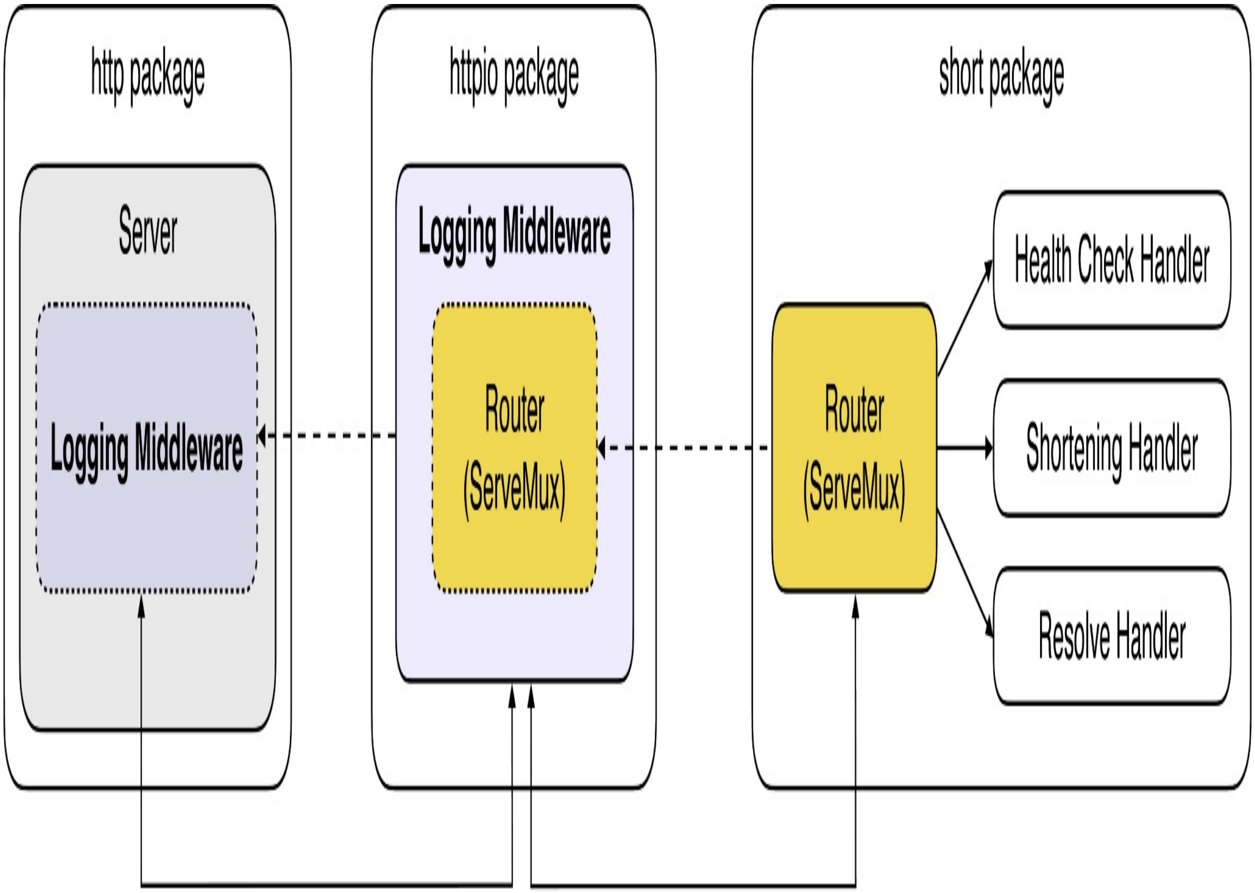
7.4 Middleware: Functional Composition

Lina, a skeptical system administrator in a tech company, was eager to implement Linkit's new URL shortener server to make the employees' lives easier. However, she was concerned about the reliability of the new server as it still needed to be battle tested. To address her concerns, Lina requested your team to add *request logging*, but only when a specific *environment variable* is set. She didn't want to convolute the log storage system.

Luckily, your team has a solution: *a logging middleware* that can be activated *before* and *after* a handler serves a request (Figure 7.7).

The logging middleware takes incoming requests, monitors them, and forwards them to the next handler (the URL shortener server's router) for further processing. It will be at the top of the food chain and monitor incoming requests and responses (as the `http.Server`'s handler).

Figure 7.7 The logging middleware monitors all requests and responses.



The Logging Middleware is assigned to http.Server as the handler and wraps the router. When http.Server receives a request, it will be forwarded to the Logging Middleware, which will route the request to the router, and the router will then route the request to the corresponding handler. This ensures that all requests pass through the Logging Middleware.

The beauty of middleware is that it can add functionalities to handlers without changing their code. Think of middleware as a chef's assistant who adds an extra flavor or functionality to a dish before serving it to the customer.

7.4.1 Implementing the logging middleware

Using Listing 7.15, let's implement the logging middleware as a helper function in the `httpio` package. To meet Lina's logging requirements, the middleware saves the start time when a request arrives and calls the next handler.

Note

Here's the `log` package's documentation: https://pkg.go.dev/log.

Once the handler finishes its job, the middleware logs a message using the Standard Library's `log` package's default—global—`Logger`, including the request duration. To achieve this, the middleware takes the next handler and returns another one—itself!

Listing 7.15: Logging middleware (./internal/httpio/log.go)

```
package httpio
// imported packages here

func LoggingMiddleware(next http.Handler) http.Handler {    #A
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.R
        start := time.Now()
        next.ServeHTTP(w, r)    #B
        end := time.Since(start)
        log.Printf("%s %s %s %v", r.Method, r.URL.Path, r.RemoteA
    })
}
```

Since you have a big brother who watches everything, in Listing 7.16, you can enable the logging middleware only when necessary by checking the `LINKIT_DEBUG` environment variable using the `os` package's `Getenv` function. This way, the middleware will be activated only if the environment variable is set.

Listing 7.16: Enabling the middleware (./cmd/shortd/shortd.go)

```
func main() {
    ...
    server := &http.Server{
        ...
        Handler: http.TimeoutHandler(shortener, timeout, "timeout
        ...
```

```
    }
    if os.Getenv("LINKIT_DEBUG") == "1" {
        server.Handler = httpio.LoggingMiddleware(server.Handler)
    }
    ...
}
```

Getting the environment variable in the `main` function is a good idea since global data can become messy if accessed from multiple places. The middleware will be enabled for the entire shortener server by setting the environment variable when running the daemon. So let's go ahead, run the server, and send it a few requests from a client program.

```
$ LINKIT_DEBUG=1 go run ./cmd/shortd
starting the server on localhost:8080
2061/07/28 15:48:13 GET /health 127.0.0.1:50451 17.209µs
2061/07/28 15:48:18 GET /r/go 127.0.0.1:50452 174.584µs
2061/07/28 15:49:55 POST /s 127.0.0.1:50462 290.459µs
```

How cool is that? You haven't touched the shortener server handlers and added a logger!

Extracting the Server's ErrorLog from Context

Lina wants to add a `"shortener: "` prefix to each log line to distinguish them. To meet this requirement, you can create a custom `Logger` with the desired prefix and a new helper function called `Log` to make it easy to use the custom logger from your handlers and middleware.

While it's a good practice to pass the `Logger` explicitly to the `Log` function, it can be a hassle to do so every time. And using the global `Logger`, as you did before in Listing 7.15, can cause issues if other parts of the code change it.

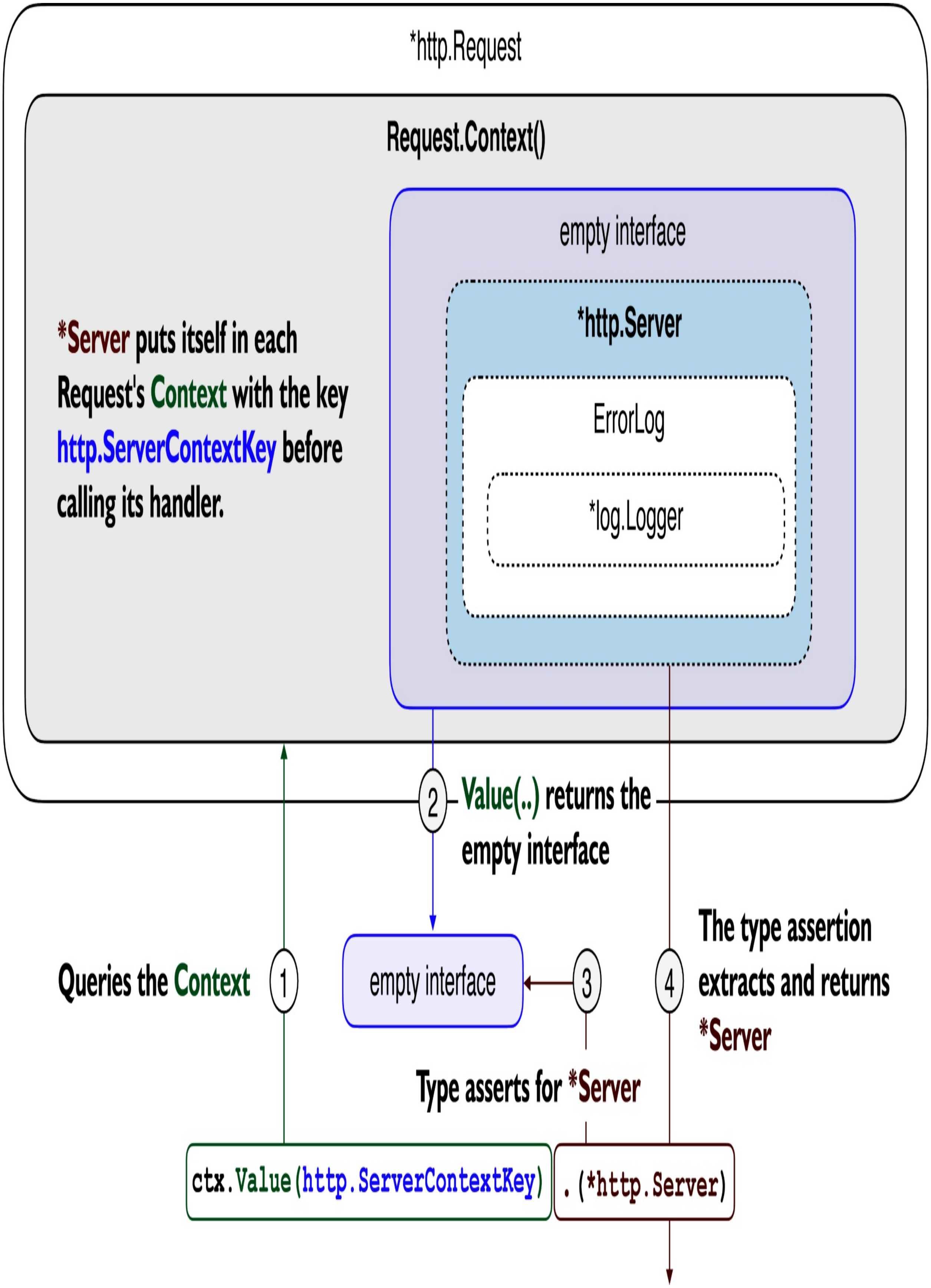
Tip

Type assertions allow you to inspect the underlying type of an interface value. It's like peeking inside a wrapped gift to see what's inside. https://go.dev/ref/spec#Type_assertions. Also, check out the official blog post for using context values: https://go.dev/blog/context.

You've got another idea! Each incoming request's `Context` includes a pointer to the `Server` that receives the request. As shown in Figure 7.8, you can retrieve this pointer using the `Value` method and passing the key, `ServerContextKey`.

However, since the method returns an empty interface that wraps the `Server` pointer, you need to use *type assertion* to *extract* the `Server` pointer from the interface. Once you have the `Server` pointer, you can get the `Server`'s `Logger`, provided it has been set.

Figure 7.8 Extracting `*Server` from `Request.Context()` using the `Value` method and type assertion.


*http.Request
Request.Context()
empty interface
*http.Server
ErrorLog
*log.Logger
*Server puts itself in each Request's Context with the key http.ServerContextKey before calling its handler.
Value(..) returns the empty interface
Queries the Context
empty interface
The type assertion extracts and returns *Server
Type asserts for *Server
ctx.Value(http.ServerContextKey)
.(*http.Server)
1
2
3
4

In Listing 7.17:

- The logging middleware receives incoming requests and passes the `Request`'s `Context` to the `Log` function.
- Then the `Log` function gets the `Server` wrapped in an empty interface from the `Context` by querying the `Value` method with `ServerContextKey`.
- Lastly, the `Log` function uses type assertion to extract the `Server` from the empty interface and log with the `Server`'s `Logger` if both the `Server` and the `Logger` exist.

Listing 7.17: Implementing a logging helper (./internal/httpio/log.go)

```
func LoggingMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.R
        ...
        Log(r.Context(), "%s %s %s %v", r.Method, r.URL.Path, r.R
    })
}

func Log(ctx context.Context, format string, args ...any) {
    s, _ := ctx.Value(http.ServerContextKey).(*http.Server)
    if s == nil || s.ErrorLog == nil {
        return
    }
    s.ErrorLog.Printf(format, args...)
}
```

Now that you have a flexible solution for getting the `Logger`, which could be a better option than using the global logger in some cases. It's time to set up a custom logger and use it throughout the `main` function. Listing 7.18 creates a custom `Logger` and sets the `Server`'s `ErrorLog` field so the `Log` function can retrieve it.

- The custom `Logger` will write log messages to the standard error stream with the prefix `"shortener: "`.
- The third argument sets the `Logger`'s output format with two flags. The first flag, `LstdFlags`, sets the default log format to include the date, time, and message.
- The second flag, `Lmsgprefix`, moves the prefix before the log message

and after the date and time to make the logs more readable and understandable.

Additional flags to customize your logger can be found in the `log` package's documentation.

Listing 7.18: Setting a logger (./cmd/shortd/shortd.go)

```
func main() {
    ... // consts
    logger := log.New(os.Stderr, "shortener: ", log.LstdFlags|log
    logger.Println("starting the server on", addr)
    ... // server init code
    if os.Getenv("LINKIT_DEBUG") == "1" {
        server.ErrorLog = logger
        server.Handler = httpio.LoggingMiddleware(server.Handler)
    }
    if err := server.ListenAndServe(); !errors.Is(err, http.ErrSe
        logger.Println("server closed unexpectedly:", err)
    }
}
```

Let's give it a try.

```
$ LINKIT_DEBUG=1 go run ./cmd/shortd
2061/07/28 15:47:07 shortener: starting the server on localhost:8
2061/07/28 15:48:13 shortener: GET /health 127.0.0.1:50451 17.209
2061/07/28 15:48:18 shortener: GET /r/go 127.0.0.1:50452 174.584µ
2061/07/28 15:49:55 shortener: POST /s 127.0.0.1:50462 290.459µs
```

Since the `LoggingMiddleware` function uses the `Log` function, incoming request logs are prefixed when you run the server with the debug environment variable enabled.

7.4.2 Handler chaining pattern

When an error occurs in a handler, it's crucial to *return* from the handler *manually*. If you don't return, the code will continue to execute and may generate incorrect, unexpected responses or security vulnerabilities. It's also tedious to add a return statement each time.

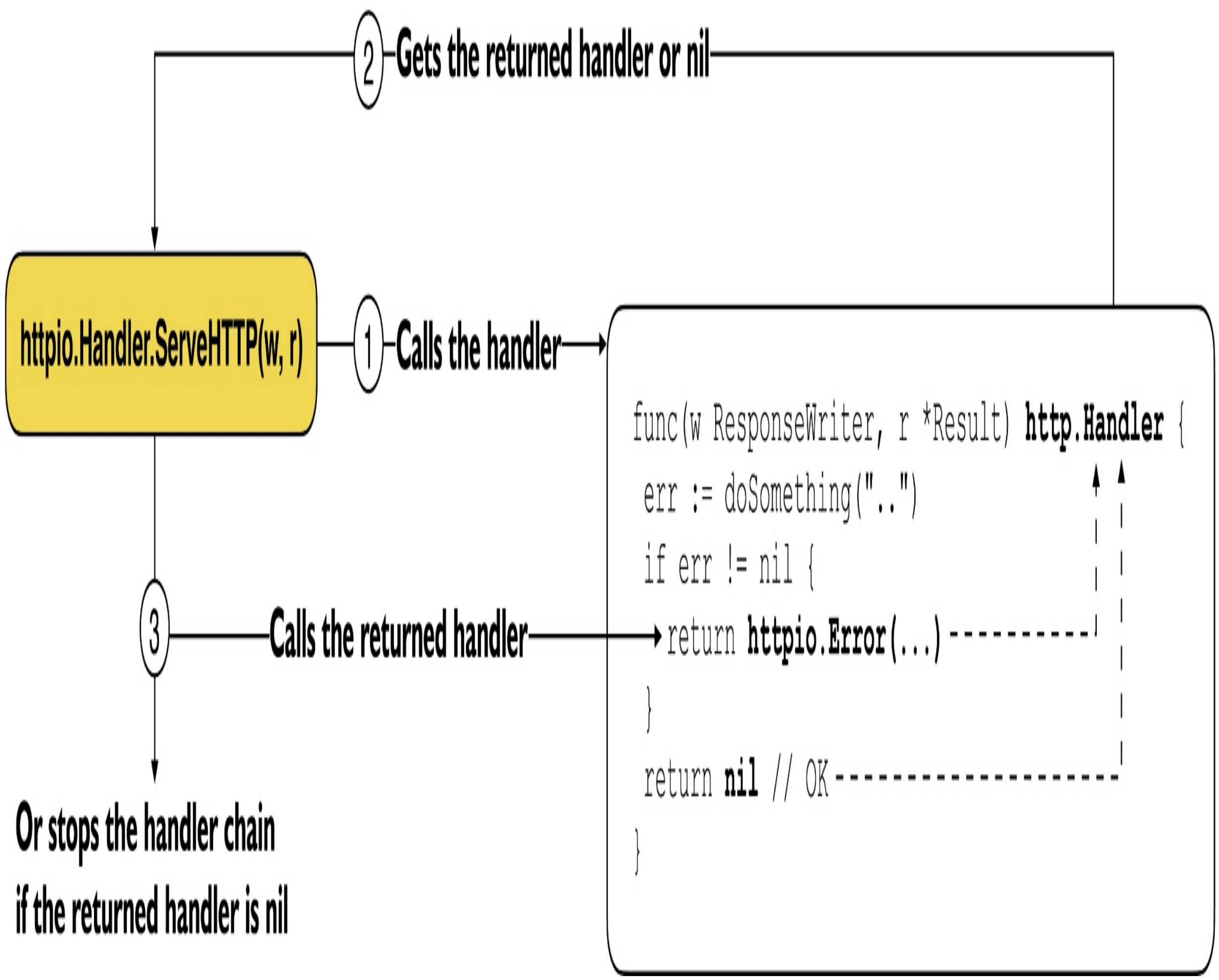
Consider the following example: you must return from the handler after an error occurs to avoid unexpected behavior.

```
func handler(w http.ResponseWriter, r *http.Request) {
    ln, err := resolve("non-existing key")
    if err != nil {
        http.Error(w, ...)
        // ...                      #A
    }
    http.Redirect(w, r, ln.uri, http.StatusFound)    #B
}
```

One solution is to *return helper handlers from your handlers*, as shown in Figure 7.9. This approach ensures you always remember to return from a handler by making handlers return other helper handlers. For instance, you can return an *error handler* to respond with an error message or a `nil` handler to stop processing the request.

Figure 7.9 Handlers act like middleware and can return a non-nil handler to continue or nil to stop handling the request.

The httpio package's Handler type has a ServeHTTP method that calls itself and then calls the returned handler if it is not nil.



By returning a helper handler from a handler, you can delegate the responsibility of handling the errors and will never forget to return.

Implementing the handler chaining pattern

Let's implement the concept you learned in Listing 7.19. The `httpio` package has a new `Handler` type with a `ServeHTTP` method. This method calls itself and then calls the returned handler to continue processing the request, provided it's not `nil`. If the returned handler is `nil`, the processing stops. Additionally, the `Error` handler is a helper for responding to the client with an error message.

Listing 7.19: Handlers act like middleware (./internal/httpio/handler.go)

```
package httpio
// imported packages here

type Handler func(w http.ResponseWriter, r *http.Request) http.Ha

func (h Handler) ServeHTTP(w http.ResponseWriter, r *http.Request
    if next := h(w, r); next != nil {    #A
        next.ServeHTTP(w, r)    #B
    }
}

func Error(code int, message string) http.HandlerFunc {    #C
    return func(w http.ResponseWriter, r *http.Request) {    #C
        http.Error(w, message, code)
    }
}
```

Another benefit of returning helper handlers from your handlers is that each helper handler can access the current `ResponseWriter` and `Request`. This allows you to use helper handlers in URL shortener handlers without passing a `ResponseWriter`.

Always returning from handlers

Now, to ensure that you always remember to return from your handlers, let's refactor them to always return a `Handler` in Listing 7.20. This requires returning a `Handler` from them and then converting them to the `httpio` package's `Handler` type when registering them.

If an error occurs, the handlers return the `Error` handler to respond with the error message to the client. On the other hand, if the operation is successful, the handlers return a `nil` handler.

Listing 7.20: Updating the URL shortener server (./short/server.go)

```
func (s *Server) registerRoutes() {
    mux := http.NewServeMux()
    mux.Handle(shorteningRoute, httpio.Handler(s.shorteningHandle
    mux.Handle(resolveRoute, httpio.Handler(s.resolveHandler))
    mux.HandleFunc(healthCheckRoute, s.healthCheckHandler) // no
```

```
    s.mux = mux
}

func (s *Server) shorteningHandler(w http.ResponseWriter, r *http
    if r.Method != http.MethodPost {
        return httpio.Error(http.StatusMethodNotAllowed, "method
    }

    var input struct {
        URL string
        Key string
    }
    err := httpio.Decode(http.MaxBytesReader(w, r.Body, 4_096), &
    if err != nil {
        return httpio.Error(http.StatusBadRequest, "cannot decode
    }

    ln := link{
        uri:      input.URL,
        shortKey: input.Key,
    }
    if err := checkLink(ln); err != nil {
        return httpio.Error(http.StatusBadRequest, err.Error())
    }
    _ = httpio.Encode(w, http.StatusCreated, map[string]any{
        "key": ln.shortKey,
    })

    return nil // success        #C
}

func (s *Server) resolveHandler(w http.ResponseWriter, r *http.Re
    key := r.URL.Path[len(resolveRoute):]

    if err := checkShortKey(key); err != nil {
        return httpio.Error(http.StatusBadRequest, err.Error())
    }
    if key == "fortesting" {
        return httpio.Error(http.StatusInternalServerError, "db a
    }
    if key != "go" {
        return httpio.Error(http.StatusNotFound, linkit.ErrNotExi
    }

    const uri = "https://go.dev"
    http.Redirect(w, r, uri, http.StatusFound)
```

```
    return nil // success        #C
}
```

The URL shortener handlers are now `httpio` handlers that return helper handlers, each returning `Error` handlers to respond with errors to clients. This practical approach reduces repetitive code and ensures you remember to return from handlers, preventing unexpected results or security vulnerabilities.

Tip

An alternative implementation can be found at https://go.dev/blog/error-handling-and-go.

7.4.3 Leaky internal errors

Lina noticed that the URL shortener server sometimes responds with an internal status code and still shows the actual error that happened, which makes her uncomfortable as she thinks the server internals shouldn't be exposed. Instead, she wants you to hide the internal errors from the clients and still log them so she can diagnose and fix the issue or send the logs to you.

```
$ curl -i localhost:8080/r/fortesting
HTTP/1.1 500 Internal Server Error
db at IP ... failed
```

To address Lina's concern, Listing 7.21 introduces an update to the `Error` handler helper that enables the handler to distinguish internal errors. If an internal error occurs, the `Error` handler logs the error and returns a generic error message to the client, keeping the complete error details in the server logs.

Listing 7.21: Logging internal errors (./internal/httpio/handler.go)

```
func Error(code int, message string) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        if code == http.StatusInternalServerError {
            Log(r.Context(), "%s: %v", r.URL.Path, message)
            message = linkit.ErrInternal.Error()
```

```
        }
        http.Error(w, message, code)
    }
}
```

The following demonstrates what it *would* look like if the resolve handler replies with an internal error (Assuming the server is running with the `LINKIT_DEBUG` on).

```
$ curl -i localhost:8080/r/fortesting
HTTP/1.1 500 Internal Server Error
internal error: please try again later or contact support
And here's the server's log:
2061/07/28 15:48:13 shortener: /r/fortesting: db at IP ... failed
```

The client receives a generic internal error message while the server logs show the error details. With these changes, Lina will be much happier!

7.4.4 Replying with JSON

As the Linkit team is getting closer to completing the URL shortener API, a customer recently complained that the returned errors were in plain text format, making it hard to extract meaningful information. Additionally, a colleague complains about having to add a `return` statement every time after responding with JSON:

```
// shortening handler
_ = httpio.Encode(w, http.StatusCreated, map[string]any{
    "key": ln.key,
})
return nil // might be unnecessary!
```

To address these concerns, the team adds a new helper handler called `JSON`, which automatically responds with JSON data in a handler when the helper is returned. The team also knows that the `Error` helper replies with plain text errors (since it uses the `http` package's `Error` function), so they update the `Error` helper to use the new JSON helper function to reply with errors in the JSON format.

In Listing 7.22, the `JSON` helper returns a `HandlerFunc` that lets you return from it in the handlers. The `Encode` helper responds with JSON; an error is logged if the encoding fails. As the `JSON` helper returns a `HandlerFunc`, you update the `Error` helper to return the `httpio` package's `Handler` type, allowing you to use the `JSON` helper.

```
Listing 7.22: Adding JSON helpers (./internal/httpio/handler.go)
func Error(code int, message string) Handler {    #A
    return func(w http.ResponseWriter, r *http.Request) http.Hand
        if code == http.StatusInternalServerError {
            Log(r.Context(), "%s: %v", r.URL.Path, message)
            message = linkit.ErrInternal.Error()
        }
        return JSON(code, map[string]string{    #A
            "error": message,
        })
    }
}

func JSON(code int, v any) http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        if err := Encode(w, code, v); err != nil {
            Log(r.Context(), "%s: JSON.Encode: %v", r.URL.Path, e
        }
    }
}
```

The `JSON` helper's addition and updating the `Error` helper will make it easier for clients to extract errors. Fortunately, you don't need to make any further changes to your handlers to reply with errors in JSON. Since all the handlers already return with the `Error` helper, they will automatically start responding with errors in the JSON format.

In Listing 7.23, you can see the updated shortener handler, which now returns the `JSON` helper when it wants to respond with JSON. The handler uses the URL shortener's data to create a new shortened link and then responds with the shortened URL in JSON format. The new `JSON` helper eliminates the need to manually add a `return` statement, saving time and making the API more manageable.

Listing 7.23: Responding with the JSON helper (./short/server.go)

```
func (s *Server) shorteningHandler(w http.ResponseWriter, r *http
    ...
    return httpio.JSON(http.StatusCreated, map[string]any{
```

```
        "key": ln.shortKey,
    })
}
```

Let's give it a try.

```
$ curl -i localhost:8080/r/skeleton
HTTP/1.1 404 Not Found
{"error":"does not exist"}
$ curl -i localhost:8080/s -d '{"url": "https://go.dev", "key": "
HTTP/1.1 201 Created
{"key":"go"}
```

Let's compare the approach in Listing 7.23 with the previous one in Listing 7.20. In the previous method, you had to encode the JSON data and then return `nil`. In the updated approach, you can directly return from the shortening handler when you want to respond with JSON, making it much simpler and easier to use.

7.4.5 Wrap up

In this section, you learned about using middleware to add extra functionality to your handlers without modifying their original code, resulting in more maintainable and flexible web programs. Then you learned how to use the `Context`'s `Value` method and type assertion to retrieve the logger stored in the request context. Additionally, you learned about the handler chaining pattern that lets you return helper handlers from your handlers to reduce mistakes and repetitive code while bringing flexibility.

7.5 Testing

Testing handlers is crucial to ensure your server behaves as expected. In the previous chapter, you learned how to test an HTTP client using the `httptest` package, which provides helper functions to test HTTP-related Go code. Testing the handlers is similar. This section will show you how to test the URL shortener server using the `httptest` package's `ResponseRecorder` type without starting a test server.

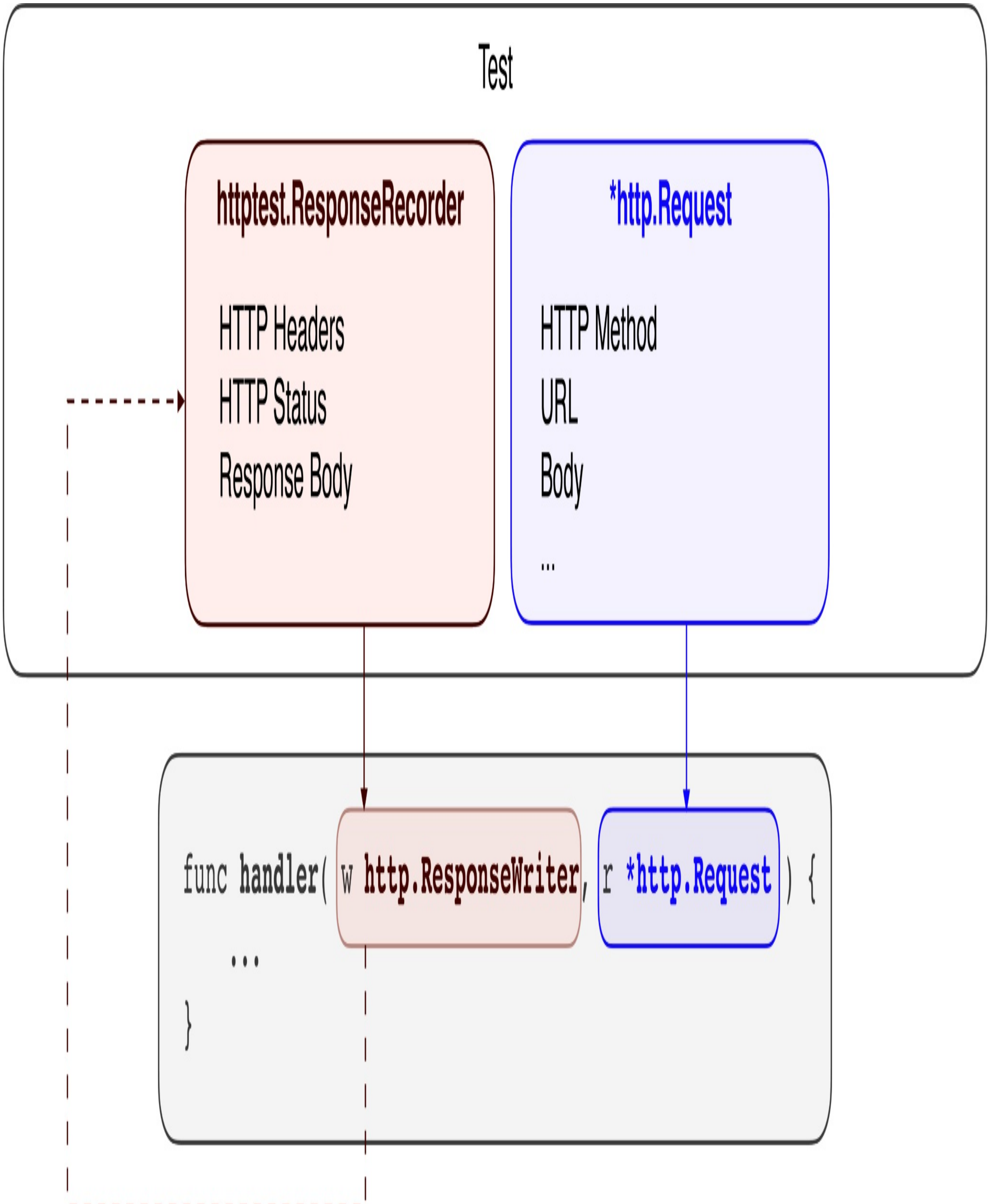
Testing handlers in isolation

Using the `httptest` package, you can create a fake `Request` and `ResponseWriter` to test your handlers in isolation. As shown in Figure 7.10, the process is straightforward.

1. Create a `ResponseRecorder` to record what the handler responds with.
2. Create a `Request` with the data the handler expects or does not expect.

Finally, pass these to the handler you want to test and use the `ResponseRecorder` methods to compare the recorded data with what you expect from the handler.

Figure 7.10 The handler receives a fake writer and request. Then `ResponseRecorder` records the handler's response.



ResponseRecorder records what the handler writes to its ResponseWriter.

In Figure 7.10, `ResponseRecorder`, a `ResponseWriter` implementation, captures and inspects the responses generated by the handler without starting a test server. It records every data the handler writes to `w` into an in-memory buffer, allowing you to verify HTTP headers, status code, and response body.

Recording handler responses with ResponseRecorder

As demonstrated in Listing 7.24, testing the shortening handler requires creating a fake `Request` and `ResponseWriter`. To do so, you first need to generate JSON data containing a URL and a short key that the handler expects to receive from the `Request`'s `Body` field as an `io.Reader`. Once you have the JSON data, you can create a fake `Request` and `ResponseWriter` to pass to the handler.

One effective way to make this data is to use a `map` value and pass it to the `Marshal` function (similar to the `Encoder` type), which converts the data to JSON and returns a `byte` slice. You can then create a new `io.Reader` from the `byte` slice using the `bytes` package's `NewReader` function and pass it as an `io.Reader` to the `NewRequest` method to make a new fake request.

Tip

A type can implicitly satisfy an interface if it has all the methods of the interface. Although the `NewReader` function returns `*bytes.Reader`, it can still be used by the `NewRequest` function as it satisfies the `io.Reader` interface, which the `NewRequest` function expects.

To create a fake `ResponseWriter`, you can use the `NewRecorder` function to create a new `ResponseRecorder` (satisfies the `ResponseWriter` interface) that records the handler's response. Finally, you can create a new shortener server and test the handler with the fake `Request` and `ResponseRecorder` values.

Listing 7.24: Testing a handler (./short/server_test.go)

```
package short
// imported packages here
```

```
func TestShortening(t *testing.T) {
    t.Parallel()

    body, err := json.Marshal(map[string]any{    #A
        "url": "https://go.dev",    #A
        "key": "go",    #A
    })    #A
    if err != nil {
        t.Fatal(err)
    }
    w := httptest.NewRecorder()    #B
    r := httptest.NewRequest(http.MethodPost, shorteningRoute, by

    srv := NewServer()
    srv.ServeHTTP(w, r)    #D

    if w.Code != http.StatusCreated {
        t.Errorf("got status code = %d, want %d", w.Code, http.St
    }
    if !strings.Contains(w.Body.String(), `"go"`) {
        t.Errorf("got body = %s\twant contains %s", w.Body.String
    }
}
```

Now that you know how to write a test for the shortening handler using a fake `Request` and `ResponseRecorder`, it's time to try it out and put your handler through its paces.

```
$ go test ./short -v
...
--- PASS: TestShortening
```

Exercise

Write a testing helper to make JSON marshaling concise and handy. It might look like this: `func jsonReader(t *testing.T, v any) *bytes.Reader { ... }`.

You now know how to test handlers without running test servers. I suggest you create another test scenario and make it as creative as you like, like testing how the handler would react to a request written in Wingdings font. Let your imagination run wild! After all, as programmers, we're used to debugging things that were supposed to work perfectly.

Handwriting the JSON

Instead of putting the JSON data in a `map` and using the `Marshal` function, you could have written the JSON string by hand and created an `io.Reader` from it using the `strings.NewReader` function as follows.

```
// makes body an io.Reader from a string value
body := strings.NewReader(`{
    "uri": "https://go.dev",
    "key": "go",
}`)
```

Although it might still be a good option for simple JSON data, this approach can be tedious to maintain and error-prone, especially with larger and more complex JSON structures.

Wrap up

Writing an HTTP server is a complex task, but Go's `http` package makes it simpler. In this chapter, you've learned how to structure, write, and test an idiomatic HTTP API using the Go Standard Library. You also learned how to utilize middleware patterns. Armed with this knowledge, you're now well-equipped to design and test your custom HTTP servers in Go.

Note

You can find a helper package designed to make testing handlers more readable and straightforward in the book's source code `./ch07/internal/httptesting`.

7.6 Exercises

1. Make the health check handler respond with JSON using the `httpio` helpers.
2. Write a middleware that allows a handler to run only if the request method matches. You can do so by registering the handlers with the new middleware function in `registerRoutes`. Then test it by only allowing `POST` requests to the shortening handler. For example,

`httpio.Allow(s.shorteningHandler, http.MethodPost)`.
3. Test the service logic functions.
4. Write additional tests for checking server errors, invalid inputs, etc.
5. Test the health and resolve handlers. Then write a test that both shortens a URL and resolves it.
6. Test all the handlers in a table test. Run each test as a subtest and in parallel. Then write a benchmark for the handlers.
7. Use the `flag` package to set server configuration—instead of constants.
8. Write an HTTP client in "`/short/client.go`" for the shortener server API. Then write a command-line tool for the client in "`./cmd/short`". You can find more details about this exercise at "`./ch07/short/client.go`".
9. Write an HTTP server API for the hit client you implemented in the previous chapter. Let clients send `POST` requests to an HTTP handler, then run the hit library in a goroutine, and respond with a short key from the handler, "`/hit`." Respond with the results when they send the short key, "`/get/key`". Use JSON encoding and decoding.

7.7 Summary

- Keep web project structures simple and pragmatic by starting with a single package and being mindful of import direction to avoid circular dependencies.
- HTTP servers listen for incoming connections and requests, while handlers manage HTTP requests and responses by generating appropriate responses, handling errors, and communicating with services.
- Services provide consistency and handle business rules to maintain a clean and maintainable codebase, while handlers manage HTTP requests and responses. Using services in handlers can keep responsibilities manageable and avoid putting too much logic in handlers.
- Timeouts can be set up to protect servers from malicious clients who may try to abuse the system with extended open connections or excessive data.
- Muxers, such as `ServeMux`, are necessary to route incoming requests to appropriate handlers based on route patterns when using the `http` package's `Server` type, which only supports a single handler.

- Embedding interfaces brings flexibility and can reduce repetitive code. Additionally, third-party routers compatible with the `Handler` interface can be incorporated.
- Middleware is an excellent way to add reusable functionality to handlers without changing them, providing increased flexibility and maintainability. The handler chaining pattern allows for the return of reusable helper handlers, such as error handling, to reduce repetitive code and maintain a clean and manageable codebase.
- Individual handlers can be tested in isolation without a test server by providing fake `ResponseWriter` and `Request` values, and using the `httptest` package's `ResponseRecorder` type, providing faster and more efficient testing of individual handlers.

8 Decoupling, Testing, and Interface Design

This chapter covers

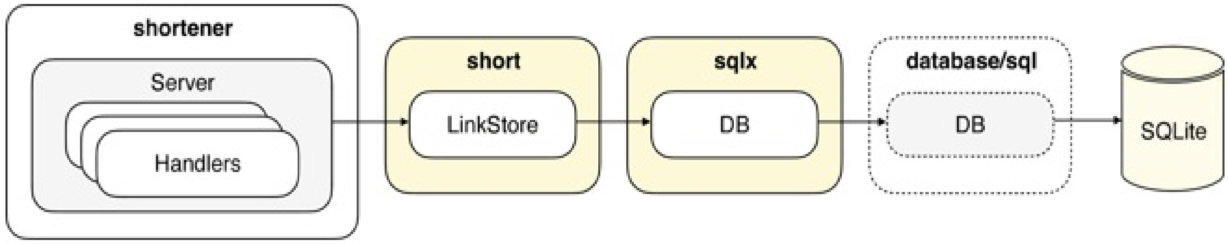
- Discovering and extending the sql package.
- Structuring code with third-party dependencies for improved maintainability.
- Designing interface types to achieve flexibility in your code.
- Testing approaches for a reliable and stable codebase.

In the previous chapter, Bite's engineering team built a URL shortener server. However, there are flaws: The server doesn't persist the shortened links, and it uses fake data! Fortunately, Lina, the product owner, has a solution. She challenges the team to integrate a storage solution to the project, allowing the links to be persisted in an SQLite database.

Figure 8.1 showcases a bird's eye view of the server with persistence.

1. The team develops a new `sqlx` package containing a `DB` type, which wraps Go's `sql` package to provide a single point of interaction for the rest of the project for interacting with SQL-like databases, such as SQLite. Wrapping the `sql` package lets the team augment it with additional functionality if needed, such as logging database methods, etc.
2. Then they add a `LinkStore` type to the `short` package to persist links. This new type uses the `sqlx` package underneath to interact with the database. And `LinkStore` employs the `sqlx.DB` type to interact with the database.

Figure 8.1 Bird's eye view of the server with persistence. The LinkStore type facilitates the storage and retrieval of links in the SQLite database, powered by the sqlx.DB type that uses Go's sql.DB type to enable interaction with the database.



This upgrade enables the server to store and retrieve shortened links, elevating its capabilities. Moreover, the team will gain valuable experience working with databases using Go's `sql` package. Throughout the chapter, you'll discover how to design, structure, and test a maintainable server with database support.

Warning

Please review the source code from the previous chapter before diving into this chapter, as this chapter builds upon it. You can find the source code for this chapter at the following link: https://github.com/inancgumus/effective-go/tree/main/ch08. Additionally, I suggest skimming the `sql` package documentation: https://pkg.go.dev/database/sql.

8.1 The database/sql package

The team is eager to add database support to the project, allowing them to create and retrieve links from the database. They discover that Go offers a package called `database/sql` (also known as the `sql` package), which enables communication with any SQL (or SQL-like, row-oriented) database.

However, the team hasn't used this package before and wants to learn more about it to interact with SQLite. In this section, you'll explore how to work with the `sql` package and create a new package called `sqlx` to provide a wrapper for interacting with the SQLite database for the rest of the project.

8.1.1 Discovering the sql package

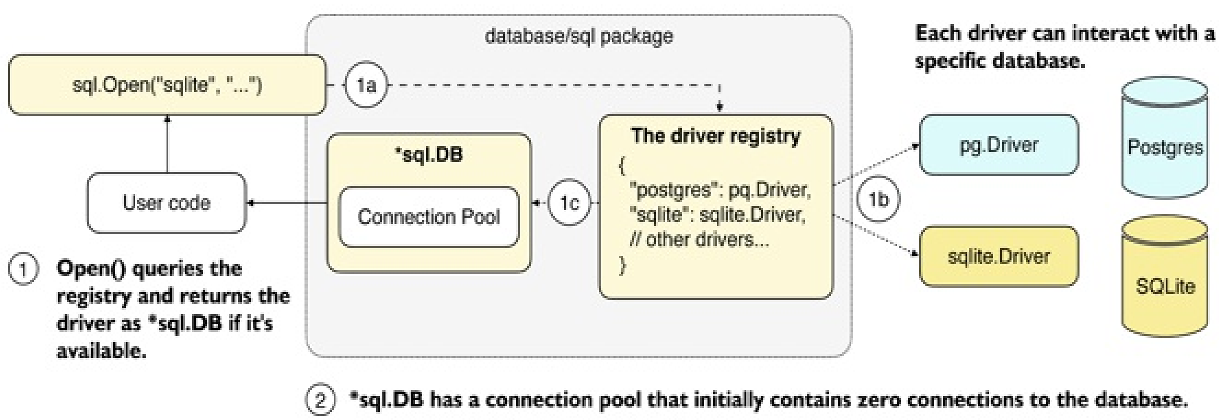
Let's get started with the `sql` package. As illustrated in Figure 8.2, the `sql` package provides a standard interface to interact with any SQL database as long as you have the necessary driver package. The drivers are often generously provided by the community and database providers. You can plug in the driver into the `sql` package to work with the desired database.

After downloading a driver, and registering it in the `sql` package, you can use the following `Open` function to get a *handle* (`*sql.DB`) and interact with a database via a connection pool.

```
func Open(driverName, dataSourceName string) (*DB, error)
// sql.Open("sqlite", "file:bite.db")
// sql.Open("mysql", "user:pass@tcp(localhost:3306)/bite")
```

The `Open` function queries the global driver *registry* and returns a handle if the requested driver is registered. Then you can use the handle to interact with the database. The first argument is the *driver's name*—usually, it's the same as the driver package's name to avoid collisions and confusion. The second one is the *driver-specific connection string*.

Figure 8.2 The sql package provides a standard interface to interact with any SQL database.



Because the `sql` package is an abstraction and only provides a standard way to communicate with SQL databases, Go does not bundle an SQL driver, and `Open` fails if you don't have the necessary driver. Since the team desires to use SQLite, next, let's download and register an SQLite driver so you can obtain a database handle to work with SQLite.

Warning

`sql.DB` is not a connection or database. Rather, it's an abstraction to seamlessly manage multiple database connections and perform auto-retries and error handling.

8.1.2 Downloading and registering a driver

The team decides to use the `modernc/sqlite` driver. The following command downloads the `sqlite` driver package with version 1.21.11 and adds it to your Go module (`go.mod`). At the time of writing this chapter, this was the latest version number.

```
$ go get modernc.org/sqlite@v1.21.1
```

Tip

For a list of additional drivers, visit https://github.com/golang/go/wiki/SQLDrivers.

Now that the driver is ready, it's time to connect to SQLite by registering the driver in the `sql` package. Listing 8.1 imports the driver with a *blank identifier* to show the compiler that you won't use the driver package's name. This is known as a *side-effect or blank import*.

Since you won't use the driver directly in your code but rather the `sql.DB` type to interact with the database, this step is necessary. If you skip it, the compiler panics with an "*imported and not used*" error, or your text editor removes the import because you're not using the driver package's functions or types in the current file.

Listing 8.1 Registering the driver (./sqlx/sqlx.go)

```
package sqlx

import (
    _ "modernc.org/sqlite" #A
)
```

Tip

Run `go mod tidy` to clean up your Go module dependency graph after saving the file. Read more about the tidy command here: https://go.dev/ref/mod#go-mod-tidy.

You created a new package called `sqlx` and registered the driver by importing it for its side effects. With the `sqlx` package, soon the team will have a single point of access to SQLite.

Let's discuss how blank importing works a little bit more. When you import a package, Go automatically executes any `init()` functions present in the package. SQL drivers commonly contain at least one `init` function, enabling them to register with the global SQL driver registry.

Tip

Avoid `init` functions in your packages as the compiler magically calls them. That can lead to code that is challenging to wrap your head around and maintain. The `sql` package uses `init` functions to maintain Go 1 backward compatibility and is stuck with them.

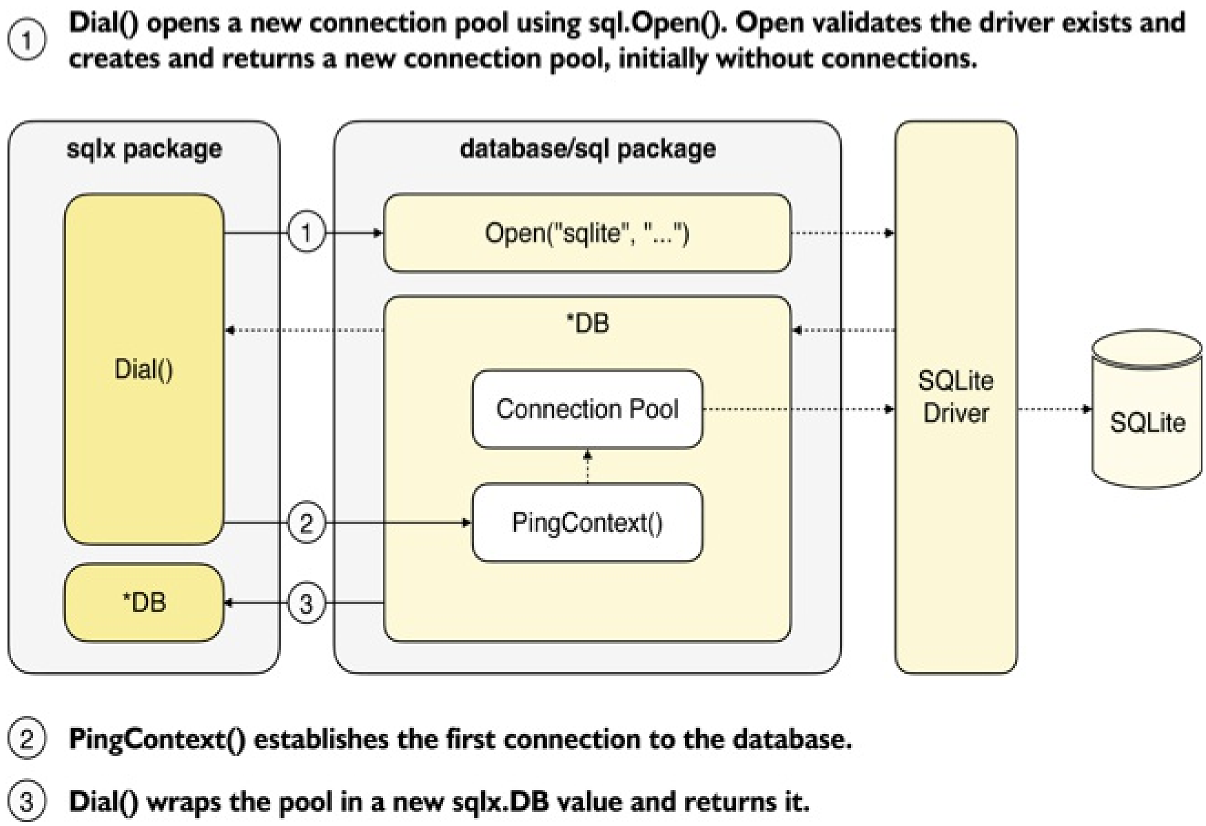
You can refer to the Go specification for further information on `init` functions: https://go.dev/ref/spec#Package_initialization (you need to scroll down a little bit).

8.1.3 Opening up a connection pool

Now that you're familiar with the `sql` package and got the driver, it's time to connect to SQLite. The team will use their `sqlx` package to establish a connection to the database. The `sqlx` package simplifies how their application interacts with the database and lets them optimize performance and add application-specific functionality when needed—such as, tracing database operations.

As you can see in Figure 8.3, the team implements the `Dial` function in the `sqlx` package to get a new database connection pool from the `sql` package. Initially, the `Dial` function gets a new empty pool with the `sql` package's `Open` function, then the function checks the validity of the connection using the pool's `PingContext` method. Lastly, the `Dial` function returns an `sqlx.DB` type by wrapping the pool handle so you can use it to persist links.

Figure 8.3 Dial opens a connection pool, pings the database, and returns the pool.



In summary, the `Dial` function returns a new `sqlx.DB` value that wraps a new connection pool (`sql.DB`). The returned pool doesn't have connections at the beginning. You can obtain a connection and validate if the database is reachable using the pool's `PingContext` method.

Optimizing the database pool

When it comes to managing connections, `sql.DB` provides several knobs to fine-tune the behavior of the connection pool for optimal performance. Within the pool, there are idle and in-use connections, and the pool checks for an idle connection when executing a database task. If none are available, a new connection is created.

Let's discuss some of the options you can use to optimize the database pool.

To manage the maximum number of connections allowed in the pool, you can use `SetMaxOpenConns` and `SetMaxIdleConns`. It's important to note that setting these values too high can lead to performance issues and inefficiencies, as idle connections consume memory and may become unusable.

`SetConnMaxLifetime` and `SetConnMaxIdleTime` are used to determine how long a connection can be reused and how long it can remain idle, respectively. Setting `MaxIdleConns` too high can result in unusable connections and wasted resources. Instead of attempting to maximize the number of connections available, it's best to optimize your connection pool settings by thoroughly testing and benchmarking them.

For further details, take a look at the Go blog post available at https://go.dev/doc/database/manage-connections.

Tip

Let me share one last tip about the connection pool. Maintain an `sql.DB` value you once obtained from the `Open` function throughout your program's lifespan since it includes a connection pool to the database. You rarely (almost never) close it since the `sql` package handles connection management automatically.

Dialing the database

Since you learned the basics, let's declare the new `DB` type and the `Dial` function in Listing 8.2 to open a new connection pool and ping the database. The function takes a driver name along with the DSN (data source name). This means the team can support multiple database drivers in the future if they ever need to use a different database.

The new `DB` type embeds `sql.DB`, allowing it to effectively behave like the actual `sql.DB` type. This means all `sql.DB` methods are conveniently available through the `sqlx.DB` type. You'll soon see how this approach streamlines code while implementing the `LinkStore` type.

Listing 8.2 Dialing the database (./sqlx/sqlx.go)

```
const DefaultDriver = "sqlite"

type DB struct {
    *sql.DB    #A
}

func Dial(ctx context.Context, driver, dsn string) (*DB, error) {
    db, err := sql.Open(driver, dsn)
    if err != nil {
        return nil, fmt.Errorf("opening database driver %q by %q:
    }
    if err := db.PingContext(ctx); err != nil {
        return nil, fmt.Errorf("pinging database: %w", err)
    }
    return &DB{DB: db}, nil
}
```

The `sqlx` package serves as a bridge between your code and SQLite. The package has a `DB` type that wraps the pool handle to later interact with the database. For the `Dial`, it establishes a *valid* connection to SQLite using the driver. In this way, the driver import happens in a single package. This lets the team take control of the driver version changes and prevent collisions in places that might import an identical driver with the same name.

Errors should tell a story

`Dial` errors tell a story: "*opening*" and "*pinging*". The usage of the present continuous tense is not accidental and allows you to figure out why and how errors occurred.

For example, consider the following errors: "*opening database driver "sqlite" by "...": sql: database is closed*". This error message tells a better story, and is less noisy and rich than the following: "*error: failed to dial the database: error: could not ping the database: sql: ...*". It's clear there is a failure because there are errors.

In short, do not unnecessarily use "*cannot*", "*failed*", "*error*", etc. in your error messages to reduce noise and improve debugging. Only use these words at the end of an error chain—the last error that doesn't wrap another error.

Testing the connection

You added a new `DB` type that wraps the `sql` package's `DB` type for consistent access to SQLite. Now that you have the necessary foundation and haven't yet integrated the `sqlx` package into the URL shortener server, writing a test is the only clean way to see if the `Dial` function works.

Listing 8.3 adds a simple test for the `Dial` function. The DSN `":memory:"` is special for SQLite and runs a new in-memory database—non-persistent, which is a great option for testing.

Listing 8.3 Testing Dial (./sqlx/sqlx_test.go)

```
package sqlx

import "testing"

func TestDial(t *testing.T) {
    db, err := Dial(context.Background(), DefaultDriver, ":memory
    if err != nil {
         t.Errorf("got err %q, want <nil>", err)
    }
    if db == nil {
         t.Error("got <nil>, want non-nil")
    }
}
```

Once you run it, you should see the test passes. And try with a different drive name and see how it fails. Let's move to applying the database schema next when you're done playing.

```
$ go test ./sqlx -v
--- PASS: TestDial
```

Exercise

Add a cancel context and cancel it before calling `Dial` to see what happens.

8.1.4 Applying the schema

Next up for the team is to define a database schema for saving short links. The schema should have the short link and the original URL. For convenience, this schema creates the table only once and won't create it if it already exists. Listing 8.4 displays a somewhat unoptimized and good enough schema for the `links` database table.

Listing 8.4 Defining the schema (./sqlx/schema.sql)

```
CREATE TABLE IF NOT EXISTS links (
    short_key VARCHAR(16) PRIMARY KEY,
    uri       TEXT NOT NULL
);
```

Since the schema is ready, let's include it in the `sqlx` package so you can execute it to have a database table. You could include the schema in a constant, but it's better to keep it in an SQL file so your text editor can highlight the syntax and even lint it.

Embedding

The question is, how do you get the schema file into the `sqlx` package?

As you can see in Listing 8.5, that's where a cool Go feature called *embedding* comes in. It lets you include a file in a variable. By the way, note that the compiler only allows embedding if you import the `embed` package for its side effects, so it knows you're going to embed files. Then you can embed the schema file into a variable using the *embed directive*. Once you've embedded the schema, you can apply it to the database using the `ExecContext` method of the `DB` type (the database connection pool handle):

```
func (db *DB) ExecContext(ctx context.Context, query string, args
```

`ExecContext` grabs a connection from the pool, executes the query, and then returns the connection to the pool. As you can infer from its signature, you can pass arguments of any type or length to the `Exec` method. Since you don't need to provide parameters to the database while running the schema, there's no need to pass any parameters for now.

Listing 8.5 Applying the schema (./sqlx/sqlx.go)

```
package sqlx

import (
    // ...
    _ "embed"     #A
)

//go:embed schema.sql #B
var schema string                #B

func Dial(ctx context.Context, driver, dsn string) (*DB, error) {
    // ...sql.Open & db.PingContext() here...
    if _, err := db.ExecContext(ctx, schema); err != nil {  #C
        return nil, fmt.Errorf("applying schema: %w", err)
    }
    return &DB{DB: db}, nil
}
#A Lets the compiler allow using the embedding directive in the f
#B Saves the schema file content into the schema variable.
#C Skipping the result value as you don't need to use it in this
```

With the embedding directive, the compiler reads the "`schema.sql`" when you compile your code and put the file's content into the `schema` variable. Then you can use it as a regular variable. Yes, you guessed it right: The Go compiler ships the file within the final binary. This approach allows you to ship a binary without any dependencies. Nice, isn't it?

Note

You can learn more about embedding at the link: https://pkg.go.dev/embed.

Wrap Up

The team took the first steps of adding database support to their URL shortener project using Go's `sql` package. They created a mediator package called `sqlx`, opened a database pool, pinged it for a new connection, embedded an SQL schema file, and executed it.

8.2 Storage service

Now that the team has successfully connected to SQLite, applied the schema, and created the `links` database table, it's time to prepare for the upcoming addition of a storage service type: `short.LinkStore` to create and retrieve links from the database.

This new type provides an abstraction for business-specific database operations used throughout the code. This helps keep the code easy to maintain and provides a clear separation of concerns between server and database operations.

Remember from the last chapter that the `short` package has two storage methods:

```
package short
func Create(ctx context.Context, ln Link) error
func Retrieve(ctx context.Context, key string) (Link, error)
```

Unlike the above functions that work with fake data, in this section, the `LinkStore` type will provide `Create` and `Retrieve` methods that can actually create and retrieve links on SQLite. The signatures of the new methods will be similar to those functions, and the handlers will be calling them.

8.2.1 High-level design of the service

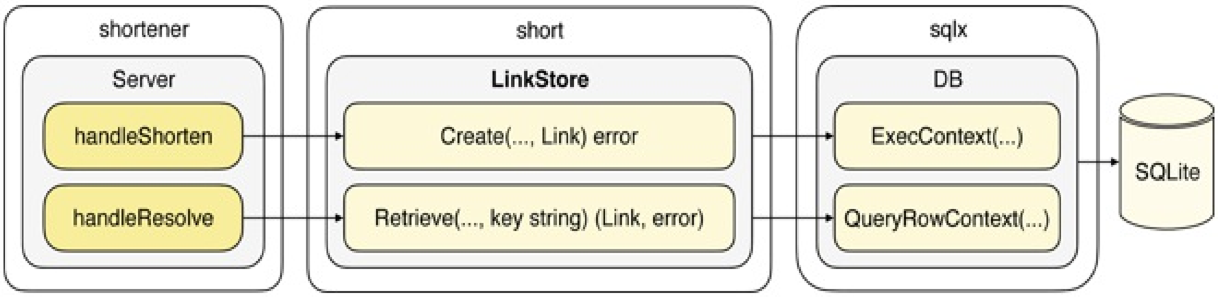
As illustrated in Figure 8.4, `sqlx.LinkStore` storage service lets the shortener server create and retrieve links using `Create` and `Retrieve` methods.

- `Create` *inserts* a link into the database using the `ExecContext` method.
- `Retrieve` uses the `QueryRowContext` method to *query* the database for a link by its short key, gets the database row data, returns the row as a short link.

Both methods use the `sqlx.DB` type to run SQL queries.

Figure 8.4 Handlers use LinkStore to create and retrieve links in the SQLite database, and

internally, LinkStore uses DB type to send queries to the database.



Recall from Listing 8.2 that `sqlx.DB` embeds the `sql.DB` type *without using a field name*, which means that all of the methods of `sql.DB` are also available for use via `sqlx.DB`. That's why you see `ExecContext` and `QueryRowContext` methods on the `sqlx.DB` type.

Once the `LinkStore` type is declared, the team will have a storage service that speaks the business language for database operations: `Create` and `Retrieve`, instead of `ExecContext` and `QueryRowContext`.

8.2.2 Creating links

Having learned about `LinkStore` for creating and retrieving short links, the team can now focus on creating links using the `Create` method. This subsection explores how to generate SQL queries and effectively insert data into the database.

Implementing the Create method

Let's get started by declaring the `LinkStore` storage service first. As you can see in Listing 8.6, `LinkStore` wraps `sqlx.DB` in a field. This way, the `Create` method can call the `ExecContext` method via the `DB` field to insert a link into the database.

Listing 8.6 Declaring the storage service type (./short/linkstore.go)

```
type LinkStore struct {
    DB *sqlx.DB
}
```

Now that you have declared the `LinkStore` type, let's move on to the `Create` method in Listing 8.7. When called, it validates the link as in the last chapter and executes an insert query with the link details.

The question marks in the query are called *placeholders* and let you substitute the link key and URL when running the query. Although the SQL query is standard SQL, the placeholders used are not.

Listing 8.7 Adding the Create method (./short/linkstore.go)

```
func (s *LinkStore) Create(ctx context.Context, ln Link) error {
    if err := validateNewLink(ln); err != nil {
        return fmt.Errorf("%w: %w", bite.ErrInvalidRequest, err)
    }
    const query = `
        INSERT INTO links (
            short_key, uri
        ) VALUES (
            ?, ?              #A
        )`
    _, err := s.DB.ExecContext(ctx, query, ln.Key, ln.URL)
    if err != nil {
        return fmt.Errorf("creating link: %w", err)
    }
    return nil
}
```

The `Create` method validates the link and inserts it into the database using the wrapped `sqlx.DB` type's `ExecContext` method, which returns the connection to the pool when it finishes its job.

Tip

To prevent SQL injection attacks, use placeholders and avoid using handcrafted SQL parameters. The placeholder syntax is database-specific. Luckily, MySQL uses the question-mark placeholders too. On the other hand, Postgres uses $1, $2, etc.

Deprecating the old Create function

Now that you have `LinkStore.Create`, you can now deprecate the previous short.Create function that uses fake data with a Deprecate notice. Labeling the function as "// Deprecated:" generates a warning in your text editor—if you're using a linter—and Go documentation, discouraging others from using it.

```
// Create will be removed soon as there is no need for it anymore
// Deprecated: Use LinkStore.Create instead.
func Create(ctx context.Context, ln Link) error { /* code here */
```

Adding a test dialer

Now that you have the Create method to persist links, let's test it. But before that, let's add a test helper to make it easier to connect to a test database. In Listing 8.8, Dial lets tests obtain an in-memory test database and close it when the calling test finishes.

Tip

Using testing.TB instead of *testing.T lets you use the Dial function both in tests and benchmarks. testing.TB is an interface that both *testing.T and *testing.B types satisfy.

Listing 8.8 Adding a test dialer (./sqlx/sqlxtest/sqlxtest.go)

```
package sqlxtest
// ...imports here...

const DefaultTestDSN = ":memory:"

func Dial(tb testing.TB) *sqlx.DB {
    tb.Helper()

    db, err := sqlx.Dial(context.Background(), sqlx.DefaultDriver
    if err != nil {
        tb.Fatalf("dialing test db: %v", err)
    }
    tb.Cleanup(func() {     #A
        if err := db.Close(); err != nil {     #A
            tb.Log("closing test db:", err)     #A
        }     #A
    })     #A
    return db
```

```
}
```

With the new test helper, connecting to a test database becomes straightforward. The `Dial` helper connects with a default driver, and it uses an in-memory database for convenience. You can add more functions here if you want to use a different DSN and driver.

Testing the Create method

Let's now create a new test in Listing 8.9 to test the `LinkStore.Create` implementation using the new test helper. The test first sets up the necessary variables and then tests the method in a subtest. You can add more subtests here.

Listing 8.9 Testing the Create method (./short/linkstore_test.go)

```
package short
// imports...

func TestLinkStore(t *testing.T) {
    var (
        ctx   = context.Background()
        store = &LinkStore{
            DB: sqlxtest.Dial(t),
        }
        link  = Link{
            Key: "go", URL: "https://go.dev",
        }
    )
    t.Run("create", func(t *testing.T) {
        err := store.Create(ctx, link)
        if err != nil {
            t.Errorf("Create() err = %q, want <nil>", err)
        }
    })
}
```

Let's check to see if the test works.

```
$ go test ./short -run=TestLinkStore -v
--- PASS: TestLinkStore/create
```

Did you notice that the `TestLinkStore` doesn't run the subtest in parallel? In this case, it's intentional, as you can add another subtest to test if `Create` can detect duplicate links.

Testing for duplicate links

You added a subtest to validate the creation of a new link. In Listing 8.10, let's add one more subtest to check whether the `Create` method permits the creation of duplicate links.

Listing 8.10 Testing if the link exists (./short/linkstore_test.go)

```
func TestLinkStore(t *testing.T) {
    // ...
    t.Run("create/exists", func(t *testing.T) {
        err := store.Create(ctx, link)
        if !errors.Is(err, bite.ErrExists) {
            t.Errorf("Create(%q) err = %q, want %q", link.Key, er
        }
    })
}
```

The test checks if the `Create` method returns `ErrExists` when adding the same link twice. Give the test a run to see if the `Create` method looks for duplicates.

```
$ go test ./short -run=TestLinkStore -v
=== RUN   TestLinkStore/create/exists
    ..: got err "creating link: ..UNIQUE constraint failed.., wan
```

The test failed because `Create` doesn't verify duplicity. Fortunately, since the schema's `short_key` column is a primary key, SQLite is designed to handle such an issue and reports an error through the driver, which `Create` then returns.

Detecting duplicate links

The previous error was clearly a unique constraint error. But in code, it can be difficult to determine whether an error is a constraint error. And parsing error messages in code can be a painful experience as they can change.

The good news is that type assertion comes to the rescue! You can extract the driver error from the error interface value returned by `Create`. Even better, the `errors` package has an `As` function that does this for you. The `As` function assigns the error to the specified error variable and returns `true` if the given error type is in the error chain. Remember, every time you wrap an error, it gets added to the error chain.

To make things easier to understand, let's use a helper function like the one shown in Listing 8.11. This function uses the `As` function to extract the driver error and save it in a variable of the same type. Once you have the driver error, you can query the database error code and return `true` if it's a primary key constraint error code.

Listing 8.11 Adding a constraint detector (./sqlx/sqlx.go)

```
package sqlx

import (
    // ...other imports here...
    "modernc.org/sqlite"        #A
    sqlite3 "modernc.org/sqlite/lib"
)

// ...the Dial function is here...

func IsPrimaryKeyViolation(err error) bool {
    var serr *sqlite.Error     #B
    if errors.As(err, &serr) {    #C
        return serr.Code() == sqlite3.SQLITE_CONSTRAINT_PRIMARYKE
    }
    return false
}
#A Uses a normal import instead of a side-effect import since the
#B Declares a variable with a driver-specific error type.
#C Extracts the driver-specific error from the given error value
#D Returns true if the driver error code is a primary key constra
```

By checking for driver-specific errors, the `sqlx` package is tightly bound to the driver package. An alternative approach would be to use transactions to query the database before inserting a row (using https://pkg.go.dev/database/sql#DB.BeginTx). This would involve more complex code and require additional round trips to the database.

The current approach strikes a nice balance. Since SQLite is already coupled to the driver, it makes sense to leverage that relationship. The `IsPrimaryKeyViolation` function is especially helpful because it contains all the driver-specific code in one place. Instead of trying to decipher cryptic SQLite errors, you can use `IsPrimaryKeyViolation` to detect duplicate links effectively. It's a much simpler and cleaner solution.

Listing 8.12 Using the constraint detector (./short/linkstore.go)

```
var ErrLinkExists = fmt.Errorf("link %w", bite.ErrExists)     #A

func (s *LinkStore) Create(ctx context.Context, ln Link) error {
    // ...
    _, err := s.DB.ExecContext(ctx, query, ln.Key, ln.URL)
    if sqlx.IsPrimaryKeyViolation(err) {
        return ErrLinkExists
    }
    // ...
}
#A A package specific error message that wraps the core error. Th
```

Let's give the test another go and see if the `Create` method is capable of detecting dups.

```
$ go test ./short -run=TestLinkStore -v
--- PASS: TestLinkStore/create/exists
```

8.2.3 Retrieving links

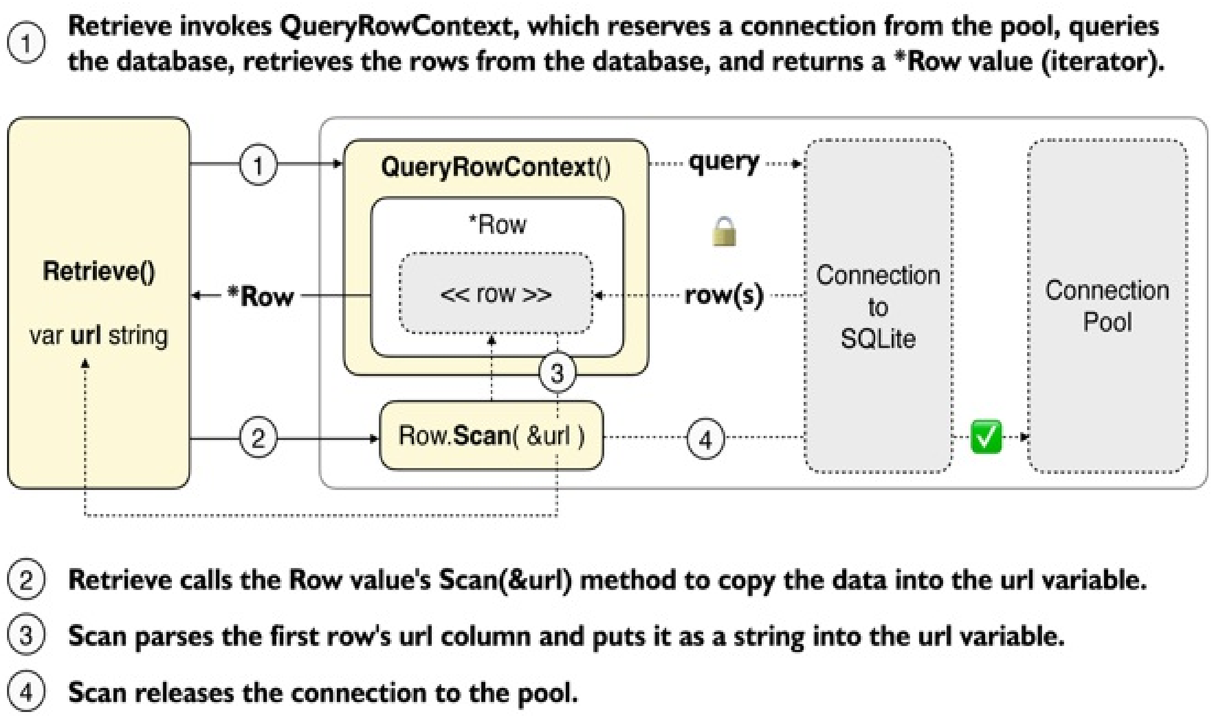
Having grasped how to insert a link into the database, let's now dive into the `Retrieve` method of the `LinkStore` type to fetch a URL from the database using a short key. Figure 8.5 offers a bird's eye view of how it operates.

When you want to pull one row from the database, `QueryRowContext` is the way to go. `Retrieve` queries the database with `QueryRowContext`, then passes a pointer to the `Scan` method, which transfers the URL from the database row into the URL variable.

Note

If your SQL query returns multiple rows, `QueryRowContext` pulls all of them from the database, and fortunately, `Scan` returns the first row and discards the rest.

Figure 8.5 Retrieve uses QueryRowContext and Scan to retrieve the short link URL from the database. QueryRowContext returns a Row value with a Scan method. Then, Scan copies the data and releases the connection back to the pool.



Essentially, `Retrieve` queries the database for a short key, and `Scan` puts the URL into a `string` variable. `Scan` gets data at the same time as releasing the connection. This boosts performance and reduces resource usage by reusing the connection for other queries.

Warning

You risk leaking database connections if you call `QueryRowContext` without `Scan`. Like `ExecContext`, `QueryRowContext` gets a connection from the connection pool. However, it reserves it for its exclusive use until `Scan` is called.

Implementing the Retrieve method

Now that you understand how you can retrieve a link URL from the database, let's dive in and implement it in Listing 8.13.

Tip

The `QueryRowContext` and `Scan` methods can take any length/type of arguments. On the other hand, for fetching multiple rows from the database, use `QueryContext`. And remember to release the returned `Rows` value, which holds a connection reference.

The `Retrieve` method gets a short URL from the database. It validates the short key, then calls `Scan` with a pointer to the `url` variable. `Scan` then parses the database result, converts it to a `string` and updates the `url` variable.

Listing 8.13 Adding the Retrieve method (./short/linkstore.go)

```
var (
    ErrLinkExists   = fmt.Errorf("link %w", bite.ErrExists)
    ErrLinkNotExist = fmt.Errorf("link %w", bite.ErrNotExist)
)

func (s *LinkStore) Retrieve(ctx context.Context, key string) (Li
    if err := validateLinkKey(key); err != nil {
        return Link{}, fmt.Errorf("%w: %w", bite.ErrInvalidReques
    }
    const query = `
        SELECT uri
        FROM links
        WHERE short_key = ?`
    var (
        url string
        err = s.DB.QueryRowContext(ctx, query, key).Scan(&url)
    )
    if errors.Is(err, sql.ErrNoRows) {     #A
        return Link{}, ErrLinkNotExist
    }
    if err != nil {
        return Link{}, fmt.Errorf("retrieving link by key %q: %w"
    }
    return Link{
        Key: key,
        URL: url,
    }, nil
```

```
}
#A ErrNoRows is a particular error type returned from sql.DB meth
```

In summary, the `Retrieve` method retrieves the short link URL from a given key. Now that you understand how it works with all the behind the scenes details, let's test it next. Before that, I'm leaving to deprecate the old `short.Retrieve` function as an exercise for you.

Testing the Retrieve method

With the `Retrieve` method under your belt, let's test its functionality by adding a test in Listing 8.14. You created a link when testing the `Create` method, so the new subtests should be straightforward. The first one retrieves a link and expects no error. The second one retrieves a bogus link and expects an error.

Listing 8.14 Testing the Retrieve method (./short/linkstore_test.go)

```
func TestLinkStore(t *testing.T) {
    // ...
    t.Run("retrieve", func(t *testing.T) {
        got, err := store.Retrieve(ctx, link.Key)
        if err != nil {
            t.Errorf("Retrieve(%q) err = %q, want <nil>", link.Ke
        }
        if got != link {
            t.Errorf("Retrieve(%q) = %#v, want %#v", link.Key, go
        }
    })
    t.Run("retrieve/not_found", func(t *testing.T) {
        _, err := store.Retrieve(ctx, "not-found")
        if !errors.Is(err, bite.ErrNotExist) {
            t.Errorf("Retrieve(%q) err = %q, want %q", link.Key,
        }
    })
}
```

What happens if you try to run the retrieve subtests before the create tests? Well, the truth is, you can't. The retrieve subtests depend on the create tests, so you have to run the create tests first. Nonetheless, this is an acceptable tradeoff.

Testing against the real thing

In this section, you tested the storage code against an actual database instead of using mocking. This lets you truly test the actual behavior.

A good example is the Go packaging server, which utilizes a test database pool to test the entire project against a real PostgreSQL database. Visit if you're interested: https://github.com/golang/pkgsite.

Docker also offers a convenient way to run an actual test database that's predictable (or "hermetic" in Google's terms). Look at the test-container project for more details to predictably test against an actual database. Visit for more: https://golang.testcontainers.org.

8.2.4 Extending the sql package

In the previous section, the team implemented `short.LinkStore`'s `Create` and `Retrieve` methods to save and retrieve links from the database. However, just when they thought they were done, product owner Lina had yet another request: she wanted URLs to be saved in the database with *Base64 encoding*. Although the team couldn't quite understand the reasoning behind this unusual feature, they were up for the challenge and confident they could fulfill the request using the `driver.Valuer` and `sql.Scanner` interfaces.

As shown in Figure 8.6, these interfaces allow modifications to be made to a value before writing it to or after reading it from a database column. *Each interface is specific to a single database column*, and drivers recognize them, calling their methods when reading and saving column data.

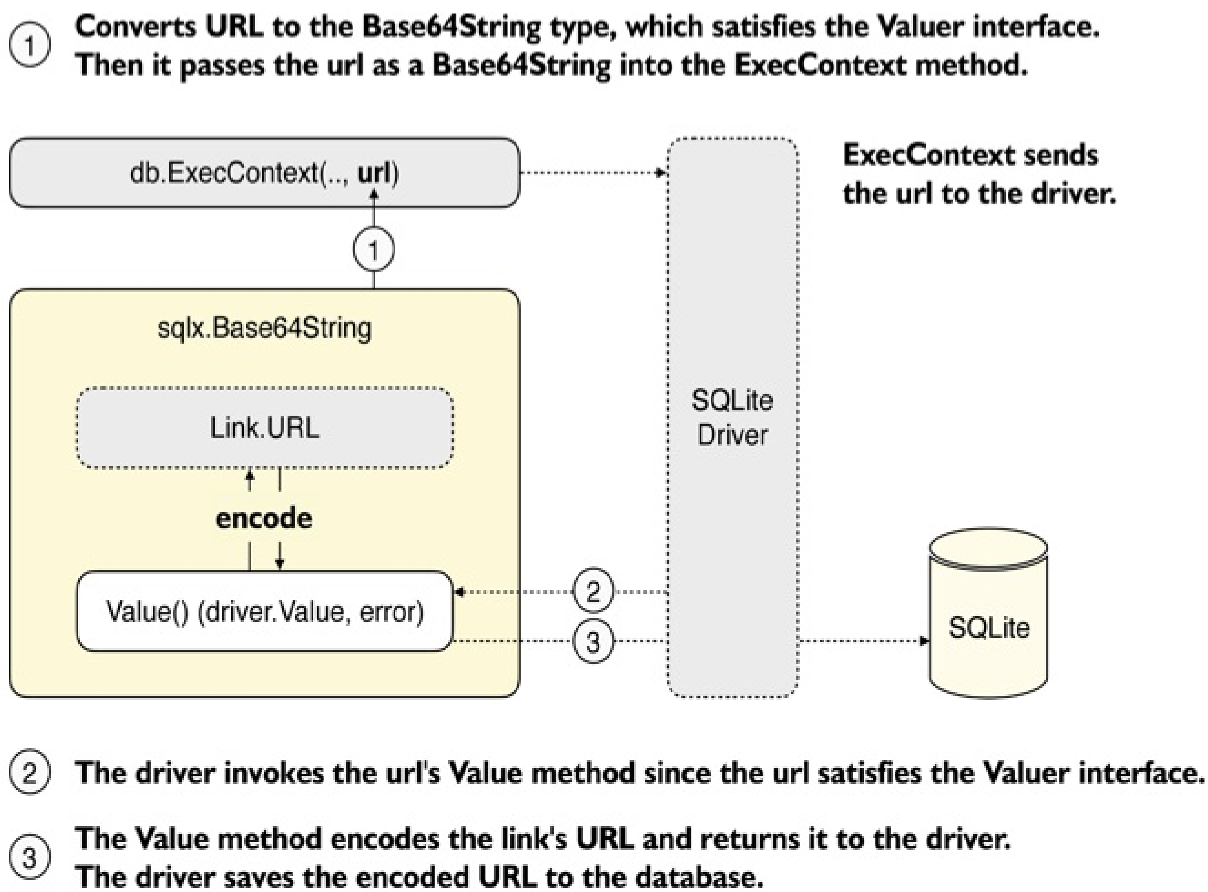
To satisfy Lina's request, the team can implement the `database/sql/driver` package's `Valuer` interface for the `uri` column to modify the URL value before sending it to the database. The `Valuer` interface has a single `Value` method. Additionally, the `database/sql` package's `Scanner` interface works in the opposite direction, allowing modifications to be made to a value after it is retrieved from a database column. It has a `Scan` method.

As Figure 8.6 demonstrates, suppose you want to encode the link's `URL` field.

In that case, you can declare another type called `Base64String` to satisfy the `Valuer` interface. Then, you can convert the `URL` field to the new type and pass the value to `ExecContext`.

`ExecContext` sends the `url` value to the driver when you do this. The driver recognizes the `url` value as a `Valuer` using type assertion and calls its `Value` method, which encodes the `url` into Base64 and returns it to the driver. Finally, the driver can tell the database to save the encoded `url`.

Figure 8.6 Using the Valuer interface to encode a value before saving it to the database. The Scanner interface is similar and works the other way around.



To sum up, the team can use the `driver` package's `Valuer` and the `sql` package's `Scanner` interfaces to encode URLs in Base64 before saving them to the database. The `Valuer` interface modifies the value before it is written to the database, while the `Scanner` interface modifies it after retrieval.

The `Base64String` type does not explicitly specify that it implements the `Scanner` and `Valuer` interfaces but does the job. This highlights the power of Go's *implicit interfaces* in customizing the behavior from a distance.

Implementing the Valuer and Scanner interfaces

Now that you understand how the `Scanner` and `Valuer` interfaces work, let's dive into implementation. To implement the encoding of URLs in Base64, the team can declare a new `string` type `Base64String` with `Value` and `Scan` methods, as shown in Listing 8.15.

- The type's `Value` method satisfies the `Valuer` interface and encodes the current value using the `base64` package's standard encoding.
- Meanwhile, its `Scan` method satisfies the `Scanner` interface and makes the given database column data a `string` using type assertion.
- Lastly, the `Scan` method overwrites the `Base64String` value after decoding the `string` value using the Base64 standard encoding (as defined in RFC 4648).

Listing 8.15 Satisfying Valuer, Scanner, and Stringer (./sqlx/encode.go)

```
package sqlx
// imports...

type Base64String string

func (s Base64String) Value() (driver.Value, error) {    #A
    dst := []byte(s)
    return base64.StdEncoding.EncodeToString(dst), nil   #A
}

func (s *Base64String) Scan(src any) error {    #B
    ss, ok := src.(string)                 #C
    if !ok {
        return fmt.Errorf("%q is %T, not string", ss, src)
    }
    dst, err := base64.StdEncoding.DecodeString(ss)
    if err != nil {
        return fmt.Errorf("decoding %q: %w", ss, err)
    }
    *s = Base64String(dst)             #D
    return nil
```

```
}

func (s Base64String) String() string {
    return string(s)
}
```

The `Base64String` type can act as a `Scanner`, `Valuer`, or `Stringer` because it satisfies these interfaces. This is made possible by Go's *implicit interfaces*, which allow a type to fulfill an interface's requirements (methods) without explicitly stating that it implements the interface.

Sending and Receiving Base64 encoded URLs

Since you have the `Base64String` type that you can use to encode and decode URL columns from the database, let's integrate it into the `LinkStore`'s `Create` and `Retrieve` methods. Listing 8.16 demonstrates how to do it.

The `Create` method first converts the `URL` to a `Base64String` type, then passes the `URL` to the `ExecContext` method. This way the `URL` is encoded before getting inserted into the database. Similarly, the `Retrieve` method passes the `url` variable as `Base64String` to the `QueryRowContext` method to get a decoded `URL`.

Listing 8.16 Using Base64String (./short/linkstore.go)

```
func (s *LinkStore) Create(ctx context.Context, ln Link) error {
    // ...
    _, err := s.DB.ExecContext(ctx, query, ln.Key, sqlx.Base64Str
    // ...
}

func (s *LinkStore) Retrieve(ctx context.Context, key string) (Li
    // ...
    var (
        url sqlx.Base64String
        err = s.DB.QueryRowContext(ctx, query, key).Scan(&url)
    )
    // ...error handling logic..
    return Link{
        Key: key,
        URL: url.String(),
    }, nil
```

```
}
```

You've successfully integrated the `Base64String` type into the `Create` and `Retrieve` methods. This means that you can now create and retrieve URLs in Base64 encoded and decoded formats. But the question is, does it actually work? Well, there's only one way to find out. Let's run the tests and see if everything's running smoothly.

```
$ go test ./short -run=TestLinkStore -v
--- PASS: TestLinkStore
```

The team has figured out how to modify column data while saving it to a database by satisfying the `Valuer` interface. On top of that, they've also learned how to change column data while retrieving it from the database by satisfying the `Scanner` interface. You can use this technique to encode and decode any column from the database. And the best part? You can apply this knowledge to so many other cases.

8.2.5 Wrap up

In this section, the team implemented the `short.LinkStore` type. This handy type wraps the `sqlx` package's `DB` type and provides a high-level storage service for the rest of the code. The `LinkStore.Create` method, using `ExecContext`, adds a `short.Link` to the database. Meanwhile, the `LinkStore.Retrieve` method uses `QueryRowContext` and `Scan` to grab a `Link` from the database. Both methods are protected against SQL injection attacks by using placeholders.

But wait, there's more! The `Base64String` function is pretty neat too. It takes advantage of the `driver.Valuer` and `sql.Scanner` interfaces to encode and decode URLs when saving and fetching them. This ensures the URLs are appropriately stored in the database and can be retrieved correctly. With all this in place, the team is ready to tackle the next challenge.

8.3 Restructuring

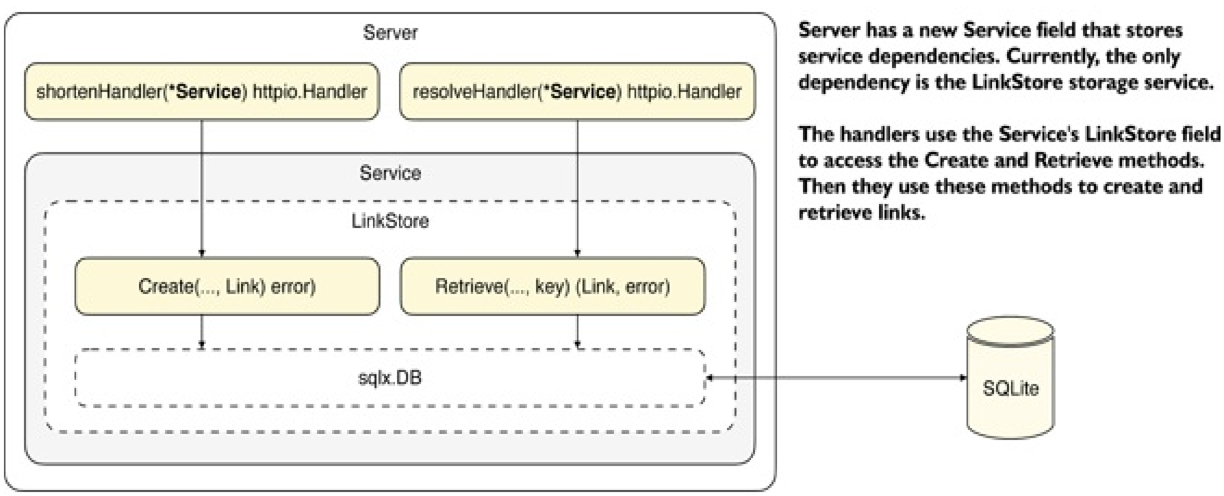
It's time to integrate the `LinkStore` storage service into the shortener server! It's like creating a beautiful symphony—each component combines perfectly to create something unique. Figure 8.7 shows the bird's eye view of the components.

First, the team introduces a new `Service` type to the `shortener` package. This type is where all the server's service dependencies are organized. It improves maintainability and represents the services the server needs, such as `LinkStore`, logging, and tracing.

When the server starts up, it passes the `Service` value to the handlers, giving them access to the storage service. This means that more service dependencies can be added to the `Service` type in the future without messing up the handler signatures. It's a win-win.

You might notice that the function signature of the handlers is different now compared to the last chapter. Since they take a `Service` type, they need to return a handler with the `httpio` package's `Handler` type so they can be registered in the server's router and continue to operate as before.

Figure 8.7 The handlers take a Service type and return a handler. They use the new Service type to access LinkStore, which lets them work with SQLite.



In summary, the handlers get a `Service` value from the `Server` type, which they use to access the `LinkStore` storage service. But how does the `Service` value with a `LinkStore` connected to the database get passed to the server? That's where the daemon program comes in. Remember, from the last chapter, `shortd` starts the URL shortening server, and this time, it initializes the server with database access.

So, what are the next steps? First, you need to implement the `Service` type. Then, restructure the `Server` and integrate the `Service` type. After that, restructure the handlers to take a `Service` and return an `httpio.Handler`. Finally, dial the database from the daemon and pass a `Service` to the `Server`.

Warning

It's a good idea to review the source code from the book's repository or the work you have done so far before moving on to the next sections that modify the code with database operations. This lets you refresh your memory and ensure that you have a good understanding of the current state of the code.

8.3.1 Implementing the Service type

Let's get started on implementing the team's plan! The first step is to create the `Service` type. Take a look at Listing 8.17 - you can see that they declare the `Service` type and add a `LinkStore` field for the link storage service—as the only service.

Listing 8.17 Implementing the Service type (./shortener/service.go)

```
package shortener
// ...imports are here...

type Service struct {
    LinkStore *short.LinkStore
}
```

The `Service` type is pretty straightforward - it just provides services to the shortener server handlers. It's a simple but crucial component in making sure everything runs smoothly.

8.3.2 Restructuring the server

Alright, time to move on to the next step in the plan - restructuring the `Server` type. As I mentioned earlier, the `Server` has a new field to access the storage service and passes the service to the handlers once it registers them

(Listing 8.18).

Listing 8.18 Integrating the Service type (./shortener/server.go)

```
type Server struct {
    http.Handler
    Service *Service
}

func (s *Server) RegisterRoutes() {
    mux := http.NewServeMux()
    mux.Handle(shorteningRoute, handleShorten(s.Service))
    mux.Handle(resolveRoute, handleResolve(s.Service))
    mux.HandleFunc(healthCheckRoute, handleHealthCheck)
    s.Handler = mux
}
```

Well, that was straightforward. Now that the `Server` has a new `Service` field that provides services to the server, and passes it to the handlers. You're almost ready to create and retrieve links. The next step is to accept the `Service` type from the handlers.

8.3.3 Restructuring to higher-order handlers

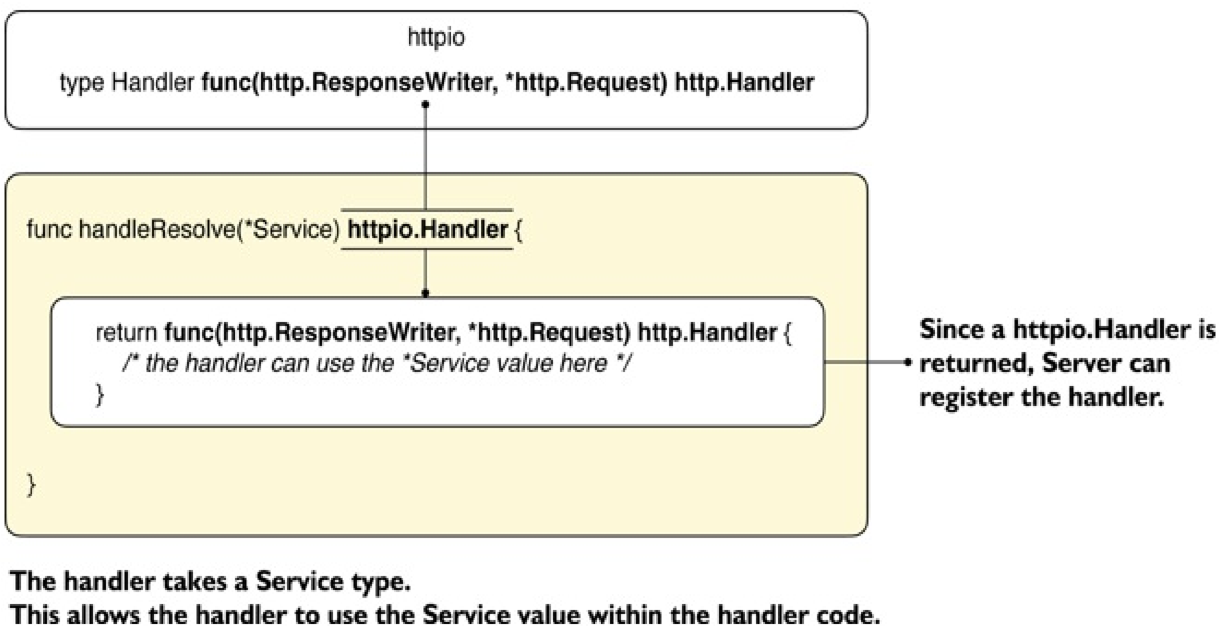
The team went ahead and combined the `Service` type with the `Server` type and passed the `Service` to the handlers. So now, it's time to let the handlers take the `Service` type as an argument so that they can use the `LinkStore` storage service through the `Service` field.

Before diving into that, let's jog your memory a bit and remember the handler signature from the last chapter. You might also recall that these handlers return helper handlers from the `httpio` package to help with processing requests. Let's check out how the handler function signature below lines up with the `httpio.Handler`'s signature in Figure 8.8.

```
func handleShorten(w http.ResponseWriter, r *http.Request) http.H
    // i.e., return httpio.Error(http.StatusConflict, err.Error()
}
```

The goal was to have the handlers use the `Service` type within their code, giving them access to the link storage service (`LinkStore`). The easiest way to pull this off is to tweak the handler signatures to accept a `Service` and return the previous handler code wrapped up in a closure as an `httpio.Handler`, just like in Figure 8.8.

Figure 8.8 The handler function signatures change to take a Service type and return an httpio.Handler. This way, the Server can register them on the muxer. And the handler can use the Service.



The handlers take a `Service` type and return an `httpio.Handler`. This way, the `Server` can register them on the muxer as `httpio` handlers. With this approach, the handlers can use the `Service` value.

Another solution could be attaching the handler functions as methods to the `Server` type. Both are fine, but keeping the functions independent lets you decouple them from the `Server` and test them directly without having to create a `Server` in the tests. Moreover, using handler functions makes what they need to work more transparent: A `Service` type.

Taking the service and returning a closure

Let's get started with restructuring the handlers in Listing 8.19. To modify the handler signatures, all you need to do is take a `Service` and return an httpio.Handler. Then, return a closure to wrap the handler code and use the storage service within the closure.

Listing 8.19 Restructuring the handlers (./shortener/server.go)

```
func handleShorten(svc *Service) httpio.Handler {
    return func(w http.ResponseWriter, r *http.Request) http.Hand
        if r.Method != http.MethodPost {
            return httpio.Error(http.StatusMethodNotAllowed, "met
        }

        var ln short.Link

        if err := httpio.Decode(http.MaxBytesReader(w, r.Body, 4_
            return httpio.Error(http.StatusBadRequest, "cannot de
        }
        if err := svc.LinkStore.Create(r.Context(), ln); err != n
            return handleError(err)
        }
        return httpio.JSON(http.StatusCreated, map[string]any{
            "key": ln.Key,
        })
    } #A
}

func handleResolve(svc *Service) httpio.Handler {
    return func(w http.ResponseWriter, r *http.Request) http.Hand
        key := r.URL.Path[len(resolveRoute):]

        ln, err := svc.LinkStore.Retrieve(r.Context(), key)  #B
        if err != nil {
            return handleError(err)
        }

        http.Redirect(w, r, ln.URL, http.StatusFound)

        return nil // success
    } #A
}
#A The handler code from the last chapter moves here.
#B Uses the link storage service to create and retrieve links in
```

The handlers now take the `Service` type and return an `httpio` handler. For example, the shortener handler can now use the storage service to create a link in SQLite. And the resolve handler can use it to retrieve a link. Almost there.

Fixing the test

After restructuring the handlers, you probably have noticed that the handler test is broken. Recall from the last chapter that you tested the handler without passing it a `Service`. Now you should create a new `Service` value and inject a test storage into it. Then you can pass the `Service` to the handler and fix the test. Let's get to it in Listing 8.20.

Listing 8.20 Restructuring the handler test (./shortener/server_test.go)

```
func TestHandleShorten(t *testing.T) {
    // test setup...
    svc := &Service{
        LinkStore: &short.LinkStore{
            DB: sqlxtest.Dial(t),
        },
    }
    handler := handleShorten(svc)
    handler.ServeHTTP(w, r)
    // assertions...
}
```

And with all that, you've successfully restructured and refactored the handlers to take a `Service` and return an `httpio.Handler`. You even fixed the shortener handler test. The next step is to run the test and see the magic happen.

```
$ go test ./shortener -run=TestHandleShorten -v
--- PASS: TestHandleShorten
```

8.3.4 Updating the daemon

You've successfully restructured the server and handlers to work with the storage service. Now, it's time to finish up the restructuring work for good by updating the shortener daemon that runs the server.

As shown in Listing 8.21, the daemon first connects to the database using a new flag (`dbDSN`), creates a new service, and then passes the service to the server. With the restructuring you did in the previous sections, the server now passes the storage service to the handlers once you register the handlers.

Listing 8.21 Updating the daemon (./cmd/shortd/shortd.go)

```
func main() {
    var (
        addr    = flag.String("addr", "localhost:8080", "server a
        timeout = flag.Duration("timeout", 10*time.Second, "serve
        dns     = flag.String("db", "file:bite.db?mode=rwc", "dat
    )
    flag.Parse()

    logger := log.New(os.Stderr, "shortener: ", log.LstdFlags|log
    logger.Println("starting the server on", *addr)

    db, err := sqlx.Dial(context.Background(), sqlx.DefaultDriver
    if err != nil {
        logger.Println("connecting to database:", err)
        return
    }
    svc := &shortener.Service{ #B
        LinkStore: &short.LinkStore{      #B
            DB: db,   #B
        },  #B
    }
    shortenerServer := &shortener.Server{
        Service: svc,   #C
    }
    shortenerServer.RegisterRoutes()
    // ...
    if err := server.ListenAndServe(); !errors.Is(err, http.ErrSe
        logger.Println("server closed unexpectedly:", err)
    }
}
#A Connects to the SQLite database with the help of the new dsn f
#B Creates a new service with the link storage service and associ
#C Injects the service into the server.
#D Remember to move this here to prevent the "no new variables" e
```

And with these changes, the team is ready to take the URL shortener server into the real world! The project is finished! Don't forget to give it a try with its new superpowers.

```
$ curl -i localhost:8080/shorten -d '{"key": "books", "url": "htt
```

```
HTTP/1.1 201 Created
{"key":"books"}
$ curl -i localhost:8080/r/books
HTTP/1.1 302 Found
Location: https://www.manning.com
```

8.4 Discovering interfaces

The product owner Lina is pleased with the team's success in building a URL shortener server with database support. Now the team faces a new challenge: some customers want to store their links in SQLite, while others prefer a cloud-based database. Unfortunately, the storage service is tightly coupled to the server, making it difficult to offer different storage options.

After much brainstorming, the team comes up with a solution. The team plans to boldly revolutionize the shortener server by decoupling the storage service via an interface. This approach unleashes new flexibility, letting the team run the server with any storage service that satisfies the interface.

Note

It's crucial to keep in mind that interfaces should be your last option, not your default. They can complicate code, making it challenging to comprehend and navigate. It's better to work with concrete types initially and consider interfaces only when necessary.

It's best to avoid premature optimization when defining interfaces. Instead, wait until they become necessary before declaring them. This way, you can avoid getting stuck with abstractions that may not be relevant later on, especially when the requirements are still unclear.

Note

As Rob Pike says: "Don't design with interfaces, discover them."

That's why the team has started working on the concrete `LinkStore` type first. Now that they discovered a need and know what methods the server needs, thanks to Go's implicit interfaces, they can do it now instead of planning for it in advance.

Structural typing and implicit interfaces

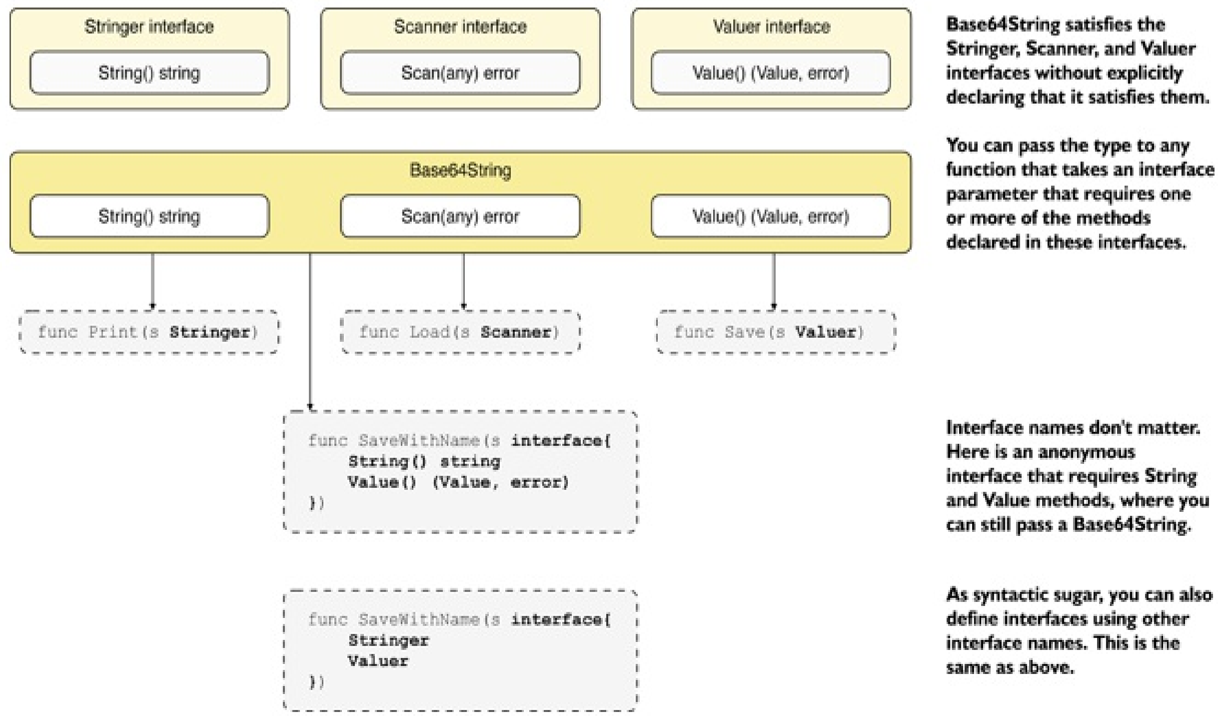
Go's structural type system makes transitioning from concrete types to interfaces effortless. Unlike some other languages, *interfaces in Go are implicitly satisfied*. You don't need to use a keyword like "*implements*" to declare that a concrete type implements a particular interface. As long as a concrete type satisfies the required methods of an interface, you can use it in any context where the interface is required.

A recent example of this can be found in Figure 8.9 of the book, which features the `Base64String` type created in Section 8.2.4. This type is implicitly a `Scanner`, `Valuer`, and `Stringer` since it implements the methods of those interfaces. This means you can pass a `Base64String` to any function or store it in a variable requiring one of those interfaces.

What's more, the interface names themselves are not required. What matters is the method names and signatures. This is the beauty of the structural type system, which allows a type to automatically satisfy an interface if it has all the methods required by that interface, regardless of the interface type name.

For instance, let's say you have a function that takes an interface called `StringMagic`, which has a `String() string` method. Even if the interface type name differs from `Stringer`, you can still pass a `Base64String` to the function because `Base64String` has a `String` method.

Figure 8.9 Base64String is a Stringer, Scanner, and Valuer without explicitly declaring.



The combination of the structural type system and implicit interfaces is compelling because it allows you to write more flexible code that can work with a broader range of types.

How to design better interfaces?

When designing interfaces, it's more useful to focus on the *behavior* or operations that types should perform, rather than their implementation details or data types. This means that instead of broad interfaces for general categories of types, it's better to declare more *specific interfaces* that define particular behaviors or operations that the types should support.

For example, you might want to create an interface for screen drivers. Instead of creating a `ScreenDriver` interface to group screen drivers by category, it's better to create interfaces like `RGBPixelColorizer`, `BrightnessSetter`, or `RefreshRateChanger` for their behavior. This allows the code to be more modular and flexible, as different types can implement these specific behaviors without needing to be part of the same general category.

The types that satisfy these interfaces don't have to be screen drivers. Various types could implement the required methods or behaviors specified by the interface. For example, a software program that manipulates images might have a type that implements the `RGBPixelColorizer` interface to manipulate the colors of individual pixels, or a home automation system might have a type that implements the `BrightnessSetter` interface to control the brightness of a set of lights.

The key is that any type that implements these interfaces must provide the expected methods or behaviors, regardless of what category of type it is.

Declare interfaces where they are used

Even though Go interfaces are powerful, they are often overused. Programmers from other languages often prematurely declare large interfaces with dozens of methods, which can lead to needlessly complex code. Later they implement those methods in a concrete type.

Tip

As Rob Pike says: "The bigger the interface, the weaker the abstraction." Avoid large interfaces that demand too much from concrete types. Define smaller, specific interfaces with only necessary methods to make it easier for concrete types to satisfy them, and to make packages more modular and maintainable. This approach makes testing, implementation swapping, and code reuse easier.

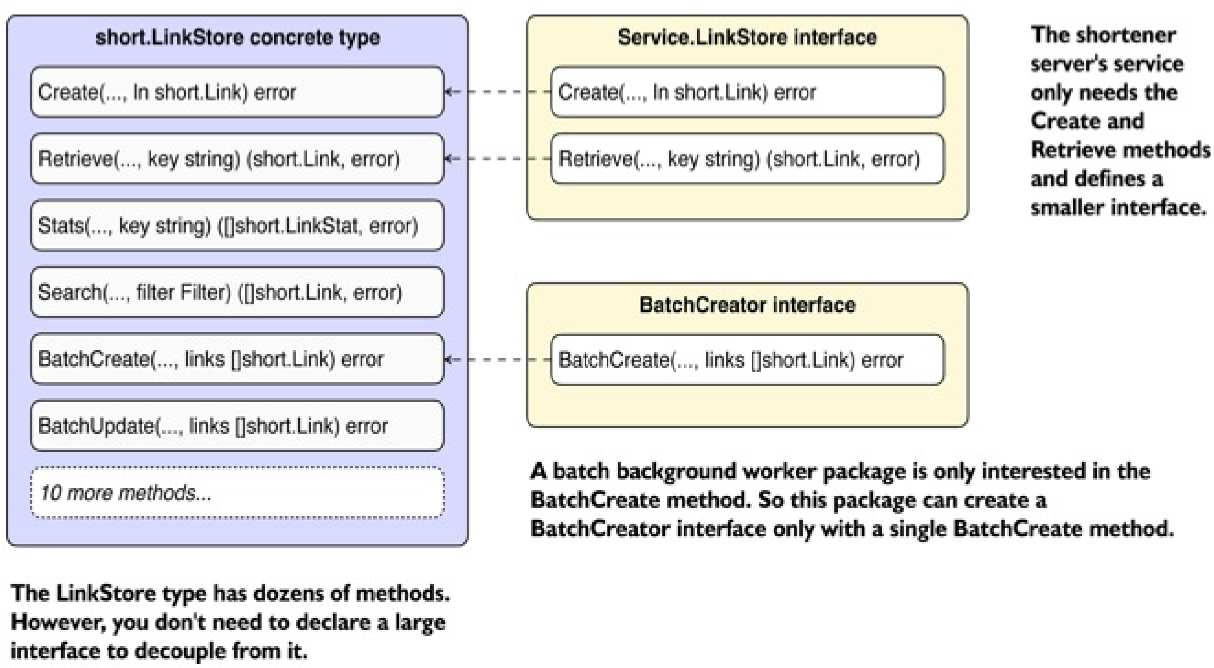
Instead, a better solution often starts with a concrete type, then lets the consumer packages decide whether to declare a small interface for their own needs. This approach makes the code more concise and reduces unnecessary complexity.

Let's take a look at an example in Figure 8.10. Currently, the `LinkStore` type has two methods: `Create` and `Retrieve`. Over time, other engineering teams added more methods for other services in the company. Thankfully, the URL shortener server only needs two of them: To create and retrieve links. Therefore, the shortener server can have an interface for what it needs from the `LinkStore` concrete type without having to include the rest of the methods.

Another example could be a batch background worker and it could create an interface for whatever it requires from the `LinkStore` type. My guess is that it would be an interface with a single method: A `BatchCreator` interface with a `BatchCreate` method.

You could go further and make every interface you declare a single-method interface, like for each handler. While it may seem logical to declare single-method interfaces for each handler, this can lead to excess noise and code complexity. Although some advocate for interfaces with a single method to make them easier to satisfy, this is not always necessary, particularly for testing purposes. Instead, one can define a struct, embed the large interface, and selectively implement only the required methods without creating additional interfaces. This approach is more user-friendly and avoids the burden of managing multiple small interfaces. Again, use your best judgment and carefully consider the trade-offs.

Figure 8.10 LinkStore has dozens of methods. The other packages can define specific and small interfaces to decouple from LinkStore.



Now that you understand the importance of defining smaller interfaces only with the method you need, let's dive into the implementation in Listing 8.22 and decouple `LinkStore` from the shortener server.

The `Service` type's `LinkStore` field has an anonymous interface with two methods: `Create` and `Retrieve`. This means that any type that has these methods can be used with the `LinkStore` field.

It just so happens that `LinkStore` satisfies this interface, so you can pass a `LinkStore` here. Now the server doesn't even have to know about the `LinkStore`'s methods and by defining an interface, the server says that: "*I need these methods and I don't care who provides them.*"

Listing 8.22 Switching to an interface (./shortener/shortener.go)

```
package short

type Service struct {
    LinkStore interface {
        Create(context.Context, short.Link) error  #A
        Retrieve(ctx context.Context, key string) (short.Link, er
    }
    // previously:
    // LinkStore *short.LinkStore
}
#A It's unnecessary to name the parameters as their types reflect
#B It's necessary to name the key here, as string doesn't reflect
```

Using a focused interface has made the code better since the server only depends on two methods that it needs rather than all the methods of `LinkStore` that it does not need. Since the `LinkStore` type satisfies the `Service.LinkStore` anonymous interface, the server and its handlers can indirectly use the `LinkStore` storage service via the anonymous interface.

You can try and see that the server can still run and serve requests, and all the tests pass. It seems like nothing has changed.

```
$ BITE_DEBUG=1 go run ./cmd/shortd
...
$ curl -i localhost:8080/shorten -d '{"key":"go", "url":"https://
HTTP/1.1 201 Created
inanc
$ curl -i localhost:8080/r/go
```

```
HTTP/1.1 302 Found
Location: https://go.dev/go
```

Alternatively, the team could have used a named interface as follows:

```
type LinkStore interface {
    Create(ctx context.Context, short.Link) error
    Retrieve(ctx context.Context, key string) (short.Link, error)
}
type Service struct {
    LinkStore LinkStore
}
```

This would have the same effect as the anonymous interface. In this case, the team chose an anonymous interface because it's more concise and easier to read and understand. There's no need to give it a name and clutter the package namespace. The critical thing to consider here is shareability - if you want others to implement the interface according to your set of rules, then it's best to declare a named interface. But if there's no need to share it, an anonymous interface is often a good choice.

Wrap up

Go's structural type system and implicit interfaces lets you start with concrete types and discover interfaces later. When defining an interface, keep it small with only the necessary methods. This approach leads to more modular and maintainable packages, easier testing, implementation swapping, and code reuse, making the code more concise and reducing unnecessary complexity.

8.5 Testing with a fake storage

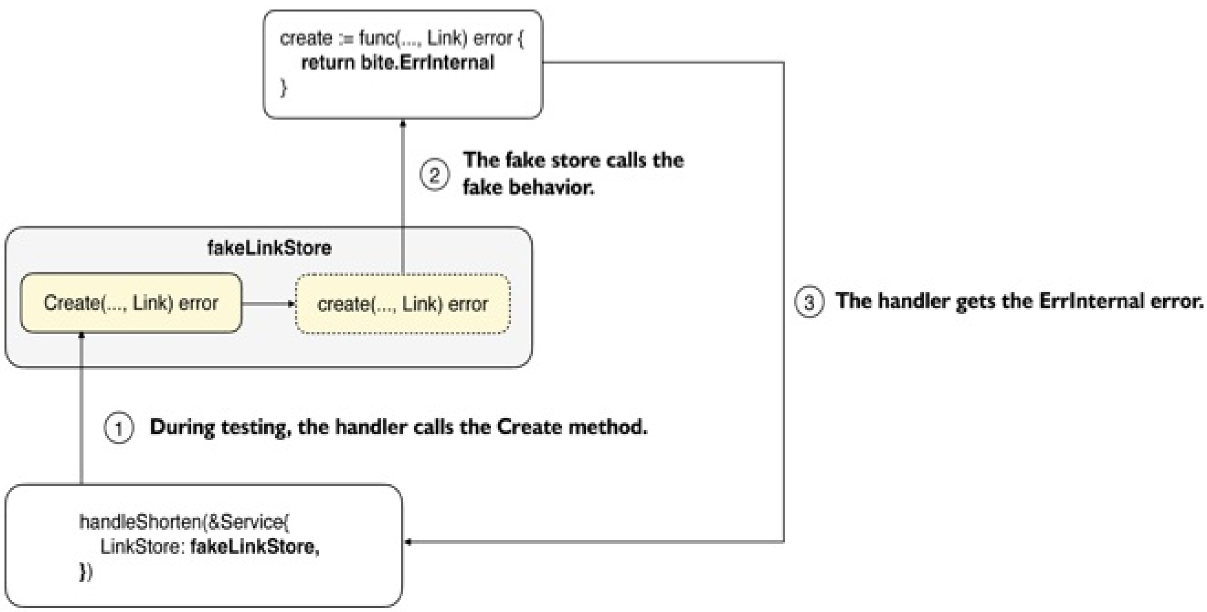
When testing in Go, small interfaces are preferred over fragile mocking frameworks often used in languages like Java or C#. The aim is to have adaptable interfaces that can swap implementations easily. Luckily, you've already created a small interface to abstract the link storage service, which makes testing easier.

The next step is to create a "*programmable*" fake link storage service that satisfies the interface, including a `Create` and `Retrieve` method, as shown in

Figure 8.11. By "programmable," I mean a fake type that can simulate different scenarios, such as returning an internal error, an "already exists" error, or no error.

By adding function value fields—also known as *hooks*—to the fake type, you can program it to return specific errors during testing. For example, you can program the `create` function field to return an internal error, and then test if the handlers can handle internal errors accurately. With this approach, you have complete control over the returned errors, allowing for a more accurate simulation of real-world error conditions.

Figure 8.11 The handler gets injected with a fake service with a fake behavior.



Once you have programmed the `fakeLinkStore` type with fake behavior, you can inject it into the shortening handler. The handler calls the fake type's `Create` method, which calls the fake `create` function. By doing this, you can see how the handler reacts when encountering an internal storage service error (without the handler realizing it is a fake).

Implementing the fake storage type

Now that you better understand the fake storage type and how it works, let's take a closer look at its implementation in Listing 8.23.

The fake type includes two hooks, `create` and `retrieve`, which allow you to program fake behaviors. It also implements the `Service.LinkStore` interface with `Create` and `Retrieve` methods, enabling you to assign a `fakeLinkStore` in the `LinkStore` field.

When the handler calls the `Create` method, the fake type returns the programmed fake behavior (create). Without programmed fake behavior, the `Create` method does not return an error for convenience. The `Retrieve` method works similarly.

Listing 8.23 Implementing the fake store (./shortener/service_test.go)

```
package shortener

type fakeLinkStore struct {
    create   func(context.Context, short.Link) error          #A
    retrieve func(context.Context, string) (short.Link, error)  #
}

func (f *fakeLinkStore) Create(ctx context.Context, ln short.Link
    if f.create == nil {     #C
        return nil      #C
    }                 #C
    return f.create(ctx, ln)   #D
}

func (f *fakeLinkStore) Retrieve(ctx context.Context, key string)
    if f.retrieve == nil {   #C
        return short.Link{}, nil     #C
    }                            #C
    return f.retrieve(ctx, key)       #D
}
```

In summary, the fake type provides a way to program the desired behavior of the storage service during testing. Using the `LinkStore` interface, you can swap out the real storage service with the fake one and test the handler's behavior under different scenarios.

Testing for an internal error using the fake storage

Now that the fake store is ready let's test the handler's reaction to an internal error from the storage using the fake store. The new test function in Listing 8.24 is akin to the current test. The distinction is that it creates and injects a fake behavior into the handler instead of using a database.

Once the test starts, the handler calls the fake store's `Create` method and receives the internal error from the fake `create` function. The test confirms that the handler returns a 500 (Internal Error) status code and an error message containing the internal error.

Listing 8.24 Testing for the internal error (./shortener/server_test.go)

```
package shortener

func TestHandleShortenInternalError(t *testing.T) {
    t.Parallel()

    body, err := json.Marshal(map[string]any{
        "key": "go",
        "url": "https://go.dev",
    })
    if err != nil {
        t.Fatal(err)
    }
    w := httptest.NewRecorder()
    r := httptest.NewRequest(http.MethodPost, shorteningRoute, by

    create := func(context.Context, short.Link) error {  #A
        return bite.ErrInternal  #A
    } #A
    svc := &Service{
        LinkStore: &fakeLinkStore{create: create},  #B
    }
    handler := handleShorten(svc)
    handler.ServeHTTP(w, r) #C

    if w.Code != http.StatusInternalServerError {
        t.Errorf("got status code = %d, want %d", w.Code, http.St
    }
    if want := bite.ErrInternal; !strings.Contains(w.Body.String(
        t.Errorf("got body = %s\twant contains %s", w.Body.String
    }
}
#A Programs the fake behavior to return an internal error.
```

```
#B Injects the fake store to the shortener service.
#C The handler calls the store's Create method and gets an intern
```

Now that the test is ready, let's test the handler's reaction to the internal error.

```
$ go test ./shortener -run=TestHandleShortenInternalError -v
--- PASS: TestHandleShortenInternalError
```

Using the fake store, the test successfully simulated how the handler reacts when an internal error occurs in the storage. Is this the only way of testing for edge cases? Let's find out next.

Alternative ways of testing

There are alternative ways to test behavior without using fake types, such as injecting testing code into the implementation. This technique is often used in the Go Standard Library to make testing for edge cases straightforward.

For instance, instead of adding a fake type to test for internal errors, you could return an internal error from the actual `Create` method when a specific key is provided, such as "`fortesting-internalerr`". This approach allows you to test the handler's behavior when an internal error occurs and identify issues before they occur in the production environment.

```
package short

func (s *LinkStore) Create(ctx context.Context, ln Link) error {
    if ln.Key == "fortesting-internalerr" {
        return bite.ErrInternal
    }
    // code that talks to the database is here.
}
```

Tip

To ensure that the test condition in the example above is not called in production code, you can use the `testing.Testing()` function to check whether the code is running in a test environment. For example "`if testing.Testing() && ln.Key == "..." { ... }`".

Then, you could use the "real" `LinkStore` implementation in your tests. This approach lets you test the behavior under different scenarios without having to create and manage fake types. Also, using this technique is not considered heresy in Go, as it is a simple and effective way to test for edge cases.

While interfaces can be helpful for testing, they can complicate the code and make navigating and comprehension challenging. Therefore, applying the simplest solution possible to produce the desired corner behavior is essential. The orthodox "*clean way*" might not always be the clearest. In some cases, injecting testing code into the implementation may be a more straightforward and explicit approach than using interfaces.

8.6 Exercises

1. Experiment with a different SQL driver (e.g., for MySQL) and modify the code accordingly. Research available SQL drivers and modify the connection setup.
2. Create a timeout context (`context.WithTimeout`) in the `sqlx.Dial` function and pass the context to the `PingContext` and `ExecContext` methods. This ensures these database operations won't wait forever.
3. Introduce support for multiple storage backends like PostgreSQL and SQLite. Develop new implementations for each backend and test their storage functionality for compatibility and correctness. Consider creating different `LinkStore` types or using an *unexported* interface within the `LinkStore` type to switch between implementations seamlessly. You can get inspired by the design of the Go CDK at the link: https://gocloud.dev/concepts/structure.
4. Introduce data expiration for shortened links by allowing them to expire after a specified or default period. Modify the `LinkStore` type to store expiration timestamps and implement a cleanup process to periodically remove expired links from the database. Consider launching a goroutine before starting the server to handle the cleanup process. Evaluate the benefits of placing the worker in a separate package or within the `short` package.
5. Implement data compression for stored links using `Valuer` and `Scanner` interfaces. Employ data compression techniques to reduce storage space requirements for links in the database. Update the storage interface and

`LinkStore` implementation to handle link data compression and decompression. Test the compression feature to verify its effectiveness and storage space savings. Research Go data compression libraries and modify the storage interface and `LinkStore` accordingly.
6. Incorporate a `LinkStats` service into the server, which could be part of the `short` package, providing click statistics for shortened links. Consider creating a new interface with the `UpdateStats` method if needed.
7. Add your own methods to the `sqlx.DB` type and log each time `ExecContext` and `QueryContext` methods are called.
8. Conduct table tests on handlers for all potential errors using the `fakeLinkStore`.
9. Test handlers without interfaces by employing conditionals and hooks (functional values) in the `LinkStore` type. Consider modifying the `LinkStore` type to include functional values and adjust the handlers to use these values instead of an interface.
10. Utilize an environment variable to skip database tests when it's not provided. This allows others to clone your repository and run tests without needing to install a database—although this is not an issue with SQLite, as it's an embedded database.

In summary, this chapter has transformed the URL shortener server by adding database support and enabling it to store and retrieve shortened links. The team's experience working with Go's `sql` package has equipped them with the skills to design, structure, and test a maintainable server with database support. This chapter is a testament to the power of Go's simplicity and flexibility, which empowers developers to create efficient, adaptable solutions that can evolve with changing requirements.

8.7 Summary

- Isolation from infrastructure components enhances maintainability. The `sqlx` package, for instance, separates database operations from the rest of the code. Using `sqlx`, the `LinkStore` type segregates server and database operations, augmenting codebase maintainability.
- Go's `sql` package provides a universal interface to work with any SQL database using third-party drivers.

- Database drivers typically registered via side-effect importing, invoking their `init` functions. To enhance maintainability, avoid `init` functions in your packages.
- The `Open` function probes the global driver registry, returning a handle to a connection pool (`*sql.DB`) if the pertinent driver is registered. Prior to registration, driver packages must be fetched using the `go get` tool.
- The `*sql.DB` type provides numerous settings to optimize the connection pool's performance, such as `SetMaxOpenConns` and `SetMaxIdleConns`. `*sql.DB` should be preserved throughout the lifespan of a program.
- The `Dial` function yields a connection pool initially devoid of connections.
- The `PingContext` method establishes a connection to the database and verifies its authenticity.
- The `ExecContext` method retrieves a connection from the pool, executes the query, and returns the connection to the pool.
- The `QueryRowContext` method retrieves a single record from the database. The `Scan` method injects database data into a Go type and relinquishes the connection back to the pool. Not coupling `QueryRowContext` with `Scan` may lead to database connection leaks.
- Placeholders offer value substitution in SQL queries and mitigate injection attacks.
- The `Scanner` interface permits modifications to a value post retrieval from a database column, while the `Valuer` interface functions in the reverse manner.
- The embed directive allows file(s) to be incorporated into the final binary.
- Labeling an identifier as `"// Deprecated:"` prompts a deprecation warning.
- The `errors.As` function assigns the error to a designated error variable, returning true if the error type is within the error chain.
- Employing `testing.TB` instead of `*testing.T` permits a test function to be utilized in both tests and benchmarks.
- Handler functions can be decoupled from the rest of the code via higher-order functions, simplifying testing.
- Testing against real infrastructure as opposed to mocking gives an accurate measure of actual behavior.

- Go's structural type system and implicit interfaces enable starting with concrete types and subsequently discovering interfaces.
- Design interfaces around the behavior types should support, not their category.
- Smaller, focused interfaces with requisite methods are easier for concrete types to satisfy.
- Interfaces better be declared where they're used. Return concrete types, and let consumer packages declare interfaces.
- Favor focused interfaces over fragile mocking frameworks.
- Programmable hooks add flexibility to testing.
- In certain scenarios, directly injecting testing code into the implementation can be a more straightforward and explicit approach than complicating code and tests with interfaces and fake types.